Optimizing L2 Cache for Datacenter Applications

Submitted in partial fulfillment of the requirements for the degree of
Master of Science (by Research)
by

Vedant Uddhaorao Kalbande
23M2108
vedantk@cse.iitb.ac.in

Supervisor:
Prof. Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2023-2026

Thesis Approval

This thesis entitled

Optimizing L2 Cache for Datacenter Applications

by

Vedant Uddhaorao Kalbande
Roll No.: 23M2108

is approved for the degree of
Master of Science by Research in Computer Science and Engineering

Digital Signature
Biswabandan Panda (10001949)
09-May-26 12:30:57 PM

Prof. Biswabandan Panda
(Supervisor)

Digital Signature
Manas Thakur (10002039)
09-May-26 12:32:57 PM

Prof. Manas Thakur
(Internal Examiner)

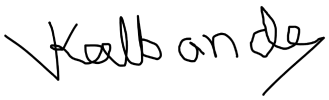
Prof. R. Govindarajan
(External Examiner)

Prof. Veeresh Deshpande
(Chairperson)

Date: May 9, 2026
Place: IIT Bombay

Declaration

I declare that this written submission represents my ideas in my own words, and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented, fabricated, or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and may also result in penal action from sources that have not been properly cited or from whom proper permission has not been obtained when required. I also declare that AI tools were used solely for language refinement and grammatical improvements of the dissertation. No generative AI-produced content has been used in the technical contributions of this dissertation.



Vedant Uddhaorao Kalbande
23M2108

Date: May 9, 2026

Nomenclature

BH	Branch History
BPU	Branch Prediction Unit
BTB	Branch Target Buffer
CIT	Criticality Indicator Table
DRAM	Dynamic Random Access Memory
FDIP	Fetch Directed Instruction Prefetching
FTQ	Fetch Target Queue
PC	Program Counter
IQ	Issue Queue
L1I	Level 1 Instruction Cache
L1D	Level 1 Data Cache
L2	Level 2 Cache
LLC	Last-Level Cache
LRU	Least Recently Used
MPKI	Misses Per Kilo Instruction

Abstract

Modern datacenter applications, characterized by large code footprints, continue to suffer from front-end CPU bottlenecks despite having a decoupled front-end. A key contributor to this bottleneck is instruction cache misses at the L2, which lead to decode starvation and degrade overall performance. Prior work, such as EMISSARY, leverages front-end criticality to identify instruction cache lines that cause decode stalls and prioritizes their retention in the L2 cache. However, we observe that only 28.32% of such critical instruction lines consistently exhibit critical behavior, while a significant fraction exhibit dynamic, context-dependent behavior.

This dissertation proposes ICARUS, a L2 cache replacement policy that improves critical instruction detection by incorporating branch history as context. In this dissertation, We observe that the reuse distance of instruction lines varies based on the branch history that led to the instruction fetch. Building on this insight, ICARUS combines criticality and reuse information to make more replacement decisions, prioritizing instruction lines that are both critical and likely to be reused.

We evaluate ICARUS across 12 datacenter applications and demonstrate that it achieves an average performance improvement of 5.6% over the baseline Tree-based Pseudo-LRU (TPLRU) policy, with gains of up to 51%. In comparison, the state-of-the-art EMISSARY policy improves performance by 2.2%. ICARUS incurs a modest storage overhead of 8.13KB for a 2MB L2 cache. We further demonstrate its robustness across a range of cache configurations, including varying L1I and L2 sizes, as well as its effectiveness in the presence of hardware prefetchers. These results highlight ICARUS as a practical and scalable solution for improving front-end performance in modern datacenter processors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the rapid expansion of cloud computing and the exponential growth of data driven by streaming platforms, online gaming, social media, and other large-scale internet services, the need for computational resources continues to rise sharply. Datacenters have become the backbone of this digital ecosystem, enabling large-scale data processing, real-time analytics, and diverse online services. With the rising demand for data and the increasing use of online services, the applications in datacenters are growing rapidly in complexity and scale, making it increasingly important to achieve higher compute efficiency. Even a marginal increase in processor performance per core leads to significant improvements in overall many-core system throughput, thereby reducing operational expenditure. This establishes performance optimisation as a pivotal objective in datacenter processor design.

1.1 Introduction to Datacenter Applications

Modern datacenter applications are becoming increasingly complex as they interact with multiple components across the system stack, including several kernel modules [1, 2], large software libraries [3], and language runtimes [4]. Consequently, their code footprint grows by approximately 30% every year [1, 3], resulting in substantial increases in both instruction and data working sets. The large instruction working set, often spanning several megabytes (MBs), typically exceeds the capacity of the level one instruction cache (L1I) and sometimes even that of the second level cache (L2), resulting in a high misses per kilo instructions (MPKI) for instruction accesses. For instance, the SPEC CPU 2017 benchmark suite [5] exhibits an average L1I MPKI of approximately 2, whereas datacenter applications have MPKI exceeding 40. This imposes enormous pressure on the cache hierarchy while fetching

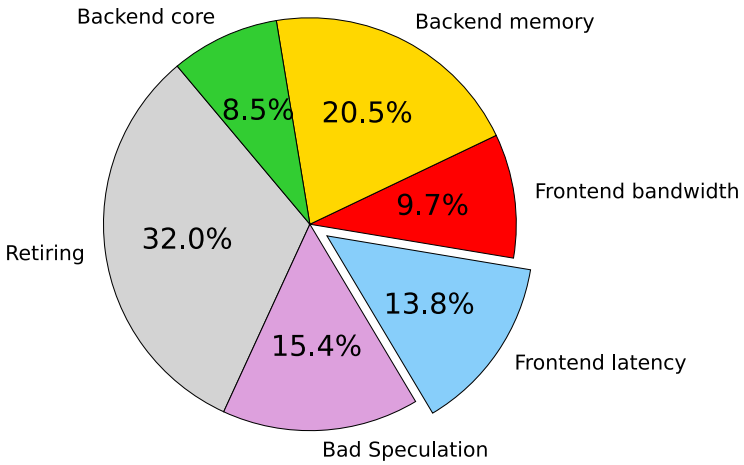


Figure 1.1: CPU performance bottleneck breakdown on a web search[1]

the instructions, causing stalls at the front-end stages of the processor pipeline. As illustrated in Figure 1.1, a breakdown of a web search workload reveals that approximately 13.8% of the total performance potential is lost due to front-end latency, primarily attributed to instruction cache misses. To mitigate this problem, several front-end optimizations have been proposed, including various instruction prefetching mechanisms [6, 7, 8, 9] and cache replacement policies [10, 11, 12], which help reduce instruction cache misses and enhance front-end performance.

1.2 The opportunity

This is evident from the industry trend, as the L1I sizes of commercial datacenter processors are not increasing significantly compared to the rate of code footprint [1, 3]. AMD uses 32KB of L1I [13], while Intel and ARM servers use 64KB of L1I [14, 15]. Mechanisms such as decoupled front-end and fetch-directed instruction prefetching (FDIP) [16, 17], as detailed in Chapter 2.1, help mitigate the latency of most L1I misses and improve the IPC by more than 40% [17] over a baseline without FDIP. However, it fails to hide latency in scenarios with long latency misses for instructions. Second-level cache (L2) misses for instruction accesses cause significant front-end bottlenecks, resulting in decode starvation(explained in detail in Chapter 2.1) and eventually leading to an empty issue queue (IQ), which completely stalls the CPU. Replacement policies for L2 that prioritize instructions [18, 12] can be essential to mitigate front-end stalls. A recent L2 replacement policy, EMISSARY [12], preserves *critical*

instruction lines that cause decode starvation and an empty issue queue, thereby improving overall front-end performance. Yet, a significant performance uplift is still possible. We quantify the performance boost that can be achieved if we have a perfect L2 for instructions (a 100% hit rate for instructions only). Figure 1.2 shows the performance uplift of different L2 replacement policies and the perfect L2 for instructions over a baseline L2 that implements a Tree-based Pseudo Least Recently Used (TPLRU) policy. For a set of datacenter applications (Table 1.1), the state-of-the-art EMISSARY provides an average speedup of 2.2%, whereas a perfect L2 for instructions provides an average speedup of 19.80%, with a maximum of 75%. Table 1.1 characterizes the evaluated benchmarks based on instruction footprint, branch predictor accuracy, critical instruction fetches, CPU stalls caused by critical fetches, and cache misses per kilo instructions (MPKI), separated by instructions and data.

A well-established replacement policy, such as Dynamic Re-Reference Interval Prediction (DRRIP) [11], improves performance by 1.6%. Dynamic Code Line Preservation (DCLIP) [18] speeds up a few datacenter applications with an average performance improvement of 0.4%. This happens primarily because DCLIP tries to keep the instruction lines at the cost of the data lines at the L2, even if not all the instruction lines cause front-end stalls. EMISSARY [12], the state-of-the-art L2 replacement policy for instructions, observes that a small percentage of instruction lines cause the majority of decode starvation, and preserves them in L2.

1.3 Contributions of This Dissertation

The goal of this dissertation work is to consider the dynamic nature of front-end criticality and reuse behavior of critical and non-critical lines while designing criticality-driven cache replacement policies for instructions. This dissertation proposes Instruction Criticality And ReUse(ICARUS), a new replacement policy for L2 that considers the pertinent observations as given below.

Observation I: EMISSARY assumes that once an instruction cache line is marked critical, it remains critical in future fetches. However, the dissertation analysis shows that instruction criticality is highly dynamic and only about 28% of critical lines retain their behavior, while the majority exhibit varying criticality due to dynamic control flow.

Observation II: EMISSARY uses a fixed threshold-based policy to retain critical lines in the L2 cache. When the number of critical lines in a set exceeds this threshold, it starts evicting them, which leads to thrashing and loss of critical lines with long reuse distances.

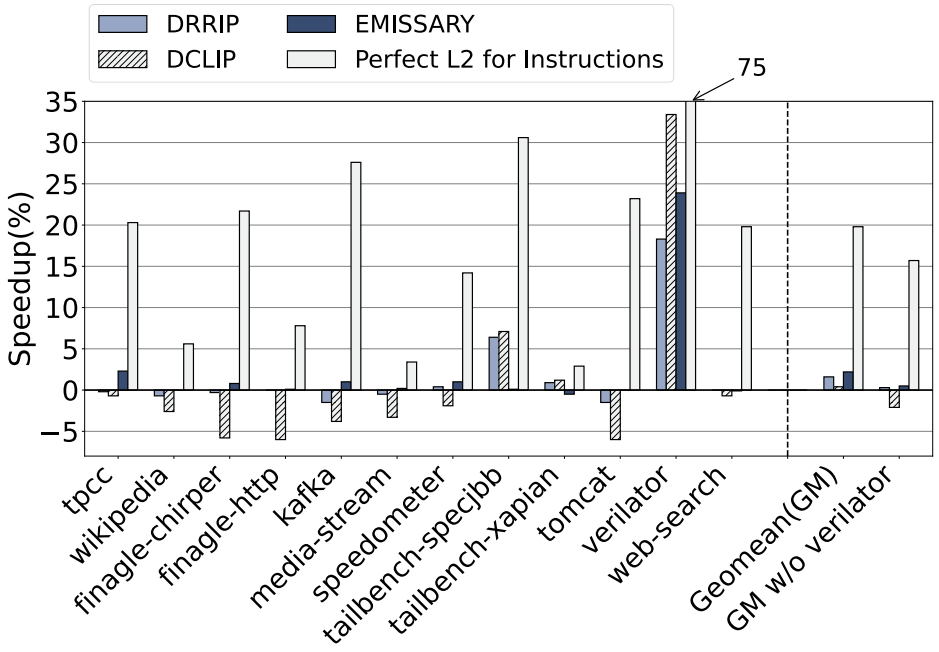


Figure 1.2: Speedup of different L2 replacement policies normalized to TPLRU across 12 datacenter applications

The key insight behind ICARUS is that the criticality of the instruction is driven by the recent control flow behavior, as the reuse of instruction lines differs based on the control flow. Datacenter applications use user-level code, libraries, virtual machines (like JVM), and OS, and the criticality of an instruction depends on the control flow jumps. The second insight is that criticality alone is not effective enough in keeping instruction cache lines of interest, as the reuse of critical and non-critical lines varies. Some critical lines exhibit short reuse behavior, and some exhibit long reuse behavior. So, there is a need for a replacement policy that uses criticality and reuse to maximize the benefits of an L2 replacement policy. We create four bins of L2 lines based on criticality and reuse and allocate the L2 space appropriately among all the bins so that the overall utility of L2 will improve in terms of L2 hits to critical instruction fetches.

The approach and contributions. First, we propose a new way to identify critical instruction lines that lead to decode starvation. We use the branch history coupled with the instruction cache line address to indicate the criticality of future instruction accesses. Our approach improves the detection of critical lines. Second, we propose different replacement priority chains for critical and non-critical instruction lines based on their reuse behavior.

Overall, we make the following contributions in the form of ICARUS: we present a case

Table 1.1: Characteristics of datacenter applications simulated for 200M instructions in their respective region of interest for a Granite Rapids-like cache hierarchy [15]: 64KB L1I, 2MB L2, and 3MB L3 per core. L2 uses the TPLRU replacement policy.

Benchmarks	Instruction Footprint (MB)	Branch Predictor Accuracy (%)	Critical Fetches (%)	Stalls due to Critical Fetches (%)	L1I MPKI	L2 Instruction MPKI	L2 Data MPKI	L3 MPKI
tpcc	2.75	96.70	2.48	23.75	33.46	1.05	1.85	1.88
wikipedia	1.14	96.74	1.58	16.14	26.53	0.28	2.20	2.35
finagle-chirper	2.91	94.74	2.96	27.68	60.39	4.87	4.60	3.29
finagle-http	1.29	94.60	3.59	30.38	70.18	0.55	1.70	1.17
kafka	1.09	98.66	1.32	8.99	8.50	1.08	4.37	4.50
media-stream	0.54	98.59	2.19	10.68	14.64	1.04	13.92	14.00
speedometer	1.45	98.40	0.92	16.26	17.38	0.45	1.58	1.82
tailbench-specjbb	0.48	99.11	1.74	5.13	12.96	5.49	6.75	7.31
tailbench-xapian	0.37	99.14	0.44	7.16	1.29	0.33	1.51	1.11
tomcat	4.15	95.48	4.44	32.89	43.82	2.75	2.59	3.51
verilator	2.22	99.44	19.89	90.69	90.23	38.50	0.16	1.07
web-search	1.50	99.33	0.34	8.35	0.72	0.26	0.39	0.65
Average	**1.66**	**97.58**	**3.49**	**23.18**	**31.68**	**4.72**	**3.47**	**3.57**

for a context-based critical instruction line detection (BHC), where we utilize the coupling of branch history with the instruction cache line address as the context. With context-based critical instruction detection, we detect the dynamic behavior of critical lines (Chapters 4.1, 4.2, and 4.3). We propose a reuse and criticality-based bins for the L2 replacement policy (BRC), which utilizes context-based critical instruction line detection and the reuse of critical and non-critical lines for the L2 replacement policy (Chapters 4.4 and 4.5). BRC enhances the residency of critical lines with long reuse, thereby improving the hit rates of critical instruction lines at L2. On average, for 12 datacenter applications evaluated using the gem5 simulator [19], ICARUS(BHC+BRC) provides a speedup of 5.6% compared to the baseline TPLRU policy. The state-of-the-art EMISSARY replacement policy provides a speedup of 2.2% (Chapter 5.2).

The organization of this dissertation. This chapter gives a brief overview of the contribution of this dissertation. The brief outline of this dissertation is as follows.

Chapter 2 provides background on modern datacenter applications, a brief about decoupled frontends in datacenter processors, and prior work on cache management policies designed specifically for datacenter processors.

Chapter 3 discusses related work on different cache management policies, instruction prefetching, cache replacement policies for other frontend structures, etc.

Chapter 4 presents ICARUS, the proposed L2 cache replacement policy. This chapter first discusses the dynamic behavior of critical instruction fetches and motivates the need for additional context using branch history. The chapter then introduces branch-history-based

criticality (BHC), criticality-and-reuse-based bins (BRC), and the main contribution, the ICARUS replacement policy, which is a combination of both BHC and BRC.

Chapter 5 presents the experimental methodology and evaluation of ICARUS. This chapter describes the simulation methodology, benchmarks, and detailed performance analysis, including speedup, instruction MPKI, and decode starvation. Further, it analyses interactions with different hardware prefetchers and sensitivity across different L1I and L2 cache sizes, BTB and CIT sizes, and cache hierarchies. This chapter also analyzes the effectiveness of ICARUS for applications with high critical instruction fetches and small L2 cache sizes. Finally, Chapter 6 concludes the dissertation and outlines directions for future work.

Chapter 2

Background

Datacenters consist of many racks of servers, and each server contains thousands of powerful computing cores. Figure 2.1 illustrates how the memory hierarchy is organized for a core. On the left, a single core is shown with its own private level 1 instruction cache (L1I), level 1 data cache (L1D), and a private Level 2 cache (L2). This core is connected to a last-level Cache(LLC) shared among multiple cores. The LLC is further connected to the main memory (DRAM), as shown on the right. These server processors often run similar types of tasks in parallel. Therefore, optimizing single-core performance by even 1% can lead to a significant overall performance improvement across the server.

2.1 Decoupled front-end

Modern datacenter processors use a decoupled front-end [16, 17], which includes a Fetch Target Queue (FTQ) placed between the branch predictor and instruction fetch stage as shown in Figure 2.2. The FTQ allows the branch predictor to work ahead of the instruction addresses sent to the L1I. It stores fetch streams generated by the branch predictor until they are either used by the L1I or squashed due to misprediction. This allows the branch predictor to run far in advance of the address currently being fetched by the cache. The decoupling enables various architecture optimizations, including multi-level branch predictor design, fetch-directed instruction prefetching, and easier pipelining of the instruction cache. This separation of the branch prediction unit and the fetch engine enables the fetch to be unaffected by the costly latency at the branch prediction unit, due to the multi-level BTB and the complex branch predictor algorithmic logic. A prefetching technique called Fetch-Directed Instruction Prefetching (FDIP), which uses the FTQ to prefetch instructions into

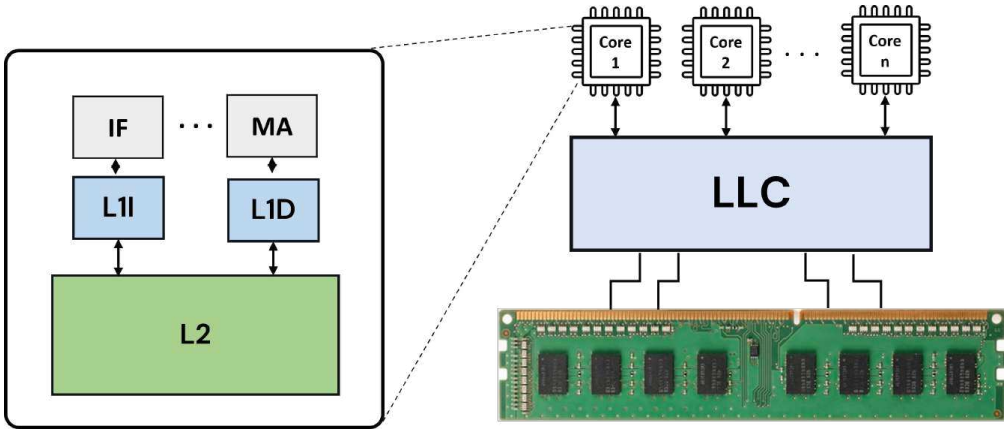


Figure 2.1: In the single processor's core(in the left), the Instruction Fetch(IF) stage is connected to a private Level 1 Instruction Cache (L1I), while the Memory Access (MA) stage is connected to a Level 1 Data Cache (L1D). Both L1I and L1D caches are connected to the Level 2 (L2) cache. The L2 cache, along with the rest of the core's cache hierarchy, is further connected to the Last-Level Cache (LLC), which is shared among multiple cores(as shown in the right figure). Finally, the LLC is connected to the main memory (DRAM).

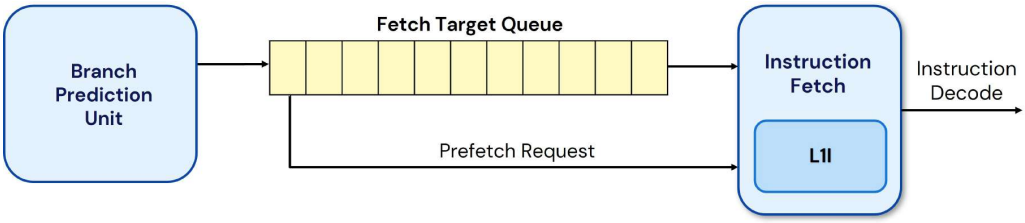


Figure 2.2: Decoupled front-end and FDIP

the L1I before they are needed. The decoupled front-end with FDIP helps tolerate the majority of front-end stalls caused by L1I misses.

2.1.1 FDIP

The decoupled architecture works such that the branch prediction unit organizes instruction flow into blocks placed into the FTQ. Each block contains multiple instructions and moves through the FTQ before being fetched. Typically, the FTQ size is around 24. This block streams through the Fetch Target Queue (FTQ), which is then consumed by the Instruction Fetch Unit. Additionally, this front-end architecture includes an effective prefetching mech-

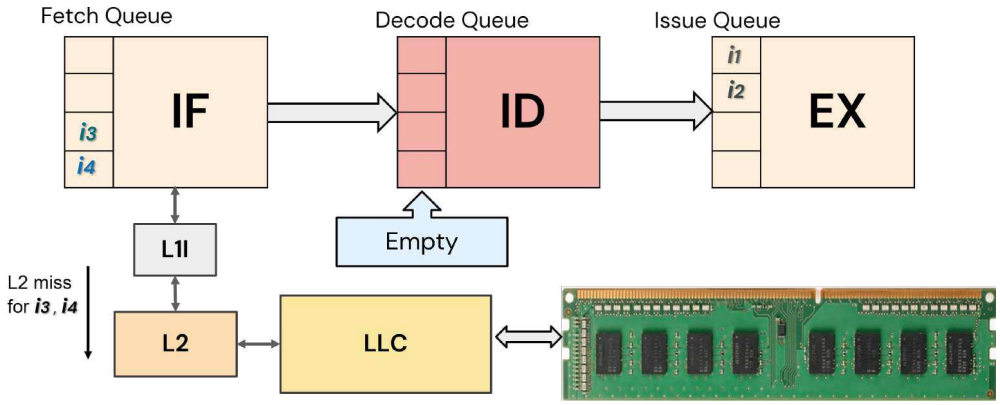


Figure 2.3: Decode Starvation

anism named Fetch Directed Instruction Prefetching (FDIP)[17, 16]. As instructions enter the FTQ, the prefetch request for that instruction block is emitted. The decoupled front end allows the processor to remain ahead of demand by fetching instructions in advance, reducing the impact of stalls caused by long-latency misses and branch mispredictions. With a decoupled front-end and FDIP, most L1I misses do not contribute to the front-end bottleneck. However, an L2 miss for an instruction fetch causes decode starvation if the decode queue is empty, leading to an empty issue queue and stalling the processor.

2.1.2 Decode starvation

Decoupled front-end and FDIP fail to hide the latency of instruction fetches in the case of decode starvation. Branch mispredictions or long-latency misses are the major causes of decode starvation. In an ideal scenario, if the processor could predict the target of every control-flow instruction perfectly, it could fetch instructions early enough to avoid stalls. However, imperfect branch predictions, even with state-of-the-art techniques[20, 21], can lead to decode starvation, where the decode stage stalls while waiting for instructions to decode. Modern processors use sophisticated techniques like decoupled fetch engines and large Branch Target Buffers (BTBs) to predict and fetch multiple instructions ahead of time. This works well when predictions are accurate, but when they go wrong, it can disrupt the flow of instructions to the decode stage, causing decode starvation.

In simple terms, decode starvation is an event where the decode stage stalls while waiting for instructions that are in flight. Please note that the processor's front-end is in order, so it can't fetch the next set of instructions until the current fetch of in-flight instructions is resolved. This issue can be further exacerbated by stalling in the issue queue, eventually

affecting the entire CPU pipeline. Figure 2.3 illustrates the event of decode starvation. When there is an L2 miss for instructions `i3` and `i4`, the decode stage stalls because the decode queue becomes empty. Please note that the previous instructions (`i1` and `i2`), fetched with them, have already been decoded and moved to the issue queue for execution, leaving no instructions in the decode queue. Since the front-end stages are in order, they cannot fetch the next set of instructions until `i3` and `i4` are inflight. As a result, an L2 miss for these instructions can stall the decode stage for hundreds of cycles in the worst case. This stalling of the decode stage is referred to as *decode starvation*.

In a decoupled architecture, although the decode stage may encounter such stalls, the fetch stage continues to work independently and is not affected by the decode stage's delay; hence, it is called decode starvation, not fetch or front-end starvation.

2.2 Replacement policies for datacenter processors

Decoupled front-end and FDIP help to mitigate the stalls at the front-end due to L1I misses, but it fails in the scenario of L2 or LLC misses, leading to decode starvation as discussed above. An L2 replacement policy can play a significant role in mitigating the front-end stalls. Code Line Preservation (CLIP) [18] preserves instruction lines over data lines at the L2. CLIP extends the 2-bit Re-Reference Interval Prediction (RRIP) [11] policy and inserts instructions with priority value two and data requests with priority value three. On future L2 hits, CLIP updates the priority of instruction lines to zero but does not change re-reference predictions for data requests. CLIP uses sampled sets to decide whether to prioritize instruction lines over data lines. EMISSARY [12] tries to keep at L2 the instruction lines that cause decode starvation. On decode starvation leading to an empty issue queue, the next instruction fetch is marked as critical. EMISSARY prioritizes critical lines over non-critical lines. Global History Reuse Predictor (GHRP) [22] is a replacement policy targeting the L1I. For eviction, GHRP prioritizes lines likely to be *dead*. GHRP is expensive to implement as it incurs a storage overhead of around 40 KB. Ripple [23] is a profile-guided software-only instruction cache replacement policy that uses profiling and program context.

2.3 EMISSARY: The state-of-the-art L2 cache replacement policy

2.3.1 EMISSARY: Overview

Modern architectures, with fetch-directed instruction prefetching (FDIP) and other aggressive front-end features, tolerate stalls due to instruction cache misses but still experience decode starvation, as discussed above. EMISSARY[12], a state-of-the-art policy for optimizing instruction caching, is designed explicitly for datacenter applications. EMISSARY supports treating instruction cache lines bimodally, distinguishing those whose misses significantly impact performance from those that do not. The factor that influences performance in this context is decode starvation, one of the significant issues, even with a decoupled front-end and fetch-directed instruction prefetching.

2.3.2 The EMISSARY Approach

Recognizing the limitations of short-lived bimodal treatment, EMISSARY policy persistently maintains a bimodal approach. Unlike other policies that differentiate only at insertion, EMISSARY treats high-priority lines differently throughout their lifespans in the cache. This is achieved by preventing insertions of low-priority lines from evicting any protected high-priority lines in the set. Each line in the L1I cache is marked with a priority bit (P = 1), which is carried over to L2 upon eviction from L1I if it caused a decode starvation.

EMISSARY cache replacement policy is bimodal, meaning it operates in two distinct modes: high and low priority. The selection of these modes is crucial for the effectiveness of the cache replacement strategy. To filter out single-use but high-priority cache lines, the "Persistent: Starvation (Decode + IQ Empty) Random EMISSARY" policy marks misses with starvation conditions as high-priority (critical) with a 1/32 probability, where the events are decode starvation, which further leads to the issue queue being empty.

2.3.3 Summary

EMISSARY, a novel L2 cache replacement policy, prioritizes cache lines based on their ability to address decode starvation(critical instruction lines), a common issue in modern architectures where FDIP fully tolerates many instruction cache misses. EMISSARY achieves performance improvements without requiring historical tracking, prefetcher coordination,

predictions, or complex calculations. However, it's worth noting that an increase in lines, leading to decode starvation, may cause interference between code and data.

2.4 The epiphany of global and local reuse distance

The reuse distance of a cache line refers to the number of cache line accesses that occur between two consecutive accesses to the same line. As it is calculated across all accesses between consecutive accesses to the same line, it should be referred to as global reuse distance. Local reuse distance of a cache line is the number of unique cache line accesses between two consecutive accesses to the same line within the same cache set. If done at L2 (which we focus on in our work), we consider cache line accesses only within the particular L2 set to calculate local reuse distance.

If the local reuse distance of a particular cache line exceeds the cache associativity, it is likely to be evicted before its next access with an LRU replacement policy. If the reuse distance is less than the cache associativity, it is likely to get a cache hit on subsequent accesses. As replacement policies make decisions per cache set, local reuse distance is a better metric than global reuse distance. Please note that state-of-the-art papers, such as EMISSARY [12], used the global reuse distance, but we use the local reuse distance throughout this study and interchangeably refer to it as the reuse distance.

Chapter 3

Related works

3.1 Cache management policies for instructions

GHRP[22] is an instruction cache replacement policy focused on minimizing the number of misses by identifying dead blocks at the L1I cache, which is orthogonal to ours. Prior work shows that GHRP fails to achieve significant performance gains in datacenter applications [23]. Ripple[23] is a software-only, profile-guided cache management technique that improves the performance of the instruction cache. A cache line that is no longer required is identified during offline analysis, and a cache line eviction instruction is inserted into the binary after the last access along all execution paths. Ripple can be applied on top of ICARUS to improve L1I performance, as they are orthogonal techniques.

Admission-Controlled Instruction Cache (ACIC) [24] incorporates a small i-Filter as done in prior works [25, 26] to separate spatial from temporal instruction accesses. However, a simple separation does not suffice, as it must also predict whether the block will continue to exhibit temporal locality after the burst of spatial locality. ACIC can be used on top of ICARUS to address instruction cache pollution. A recent work [27] proposes cooperative last-level TLB (STLB) and L2 replacement policies. The policy improves the instruction translation hit rates at the STLB at the cost of data translation misses. To compensate for that, it prioritizes data translation at the L2 level. The applications that we use do not have significant instruction translation misses at STLB. In the case of high-instruction translation MPKI, ICARUS can be used at the STLB too, preserving critical translation lines.

3.2 Instruction prefetching

ICARUS preserves instruction lines at the L2. However, instruction prefetching techniques go one step further and prefetch instructions into L1I. Priority-directed instruction prefetching (PDIP)[6] is an instruction prefetching technique that complements FDIP by issuing prefetches for only targets where FDIP struggles. As shown in Figure 5.6, ICARUS complements PDIP well. Utility-Driven fetch-directed instruction Prefetching (UDP) [28] improves the accuracy and timeliness of FDIP by prefetching instructions along the wrong path. UDP shows that dynamically tuning the FTQ depth can improve the accuracy and timeliness of instruction prefetch requests. Similar to PDIP, UDP will complement ICARUS well, as ICARUS improves L2 performance and UDP improves L1I performance. Entangled Instruction Prefetching (EIP) [29] proposed entangling-based prefetching that associates a cache miss-causing line of a variable-latency L with an access that occurs L cycles prior. Like PDIP and UDP, EIP can be seamlessly used with ICARUS.

3.3 Cache replacement for other front-end structures

A profile-guided BTB replacement policy, Thermometer [30], leverages both transient and holistic(across the entire execution) branch behavior to reduce BTB misses in datacenter workloads. It relies on offline profiling to generate replacement hints, enabling the hardware to make better eviction decisions; however, its reliance on profiling may limit its adaptability to dynamic execution behavior. FURBYS [31] is a profile-guided micro-op cache replacement policy that addresses the limitations of conventional policies in handling variable miss costs and partial hits. Guided by an offline near-optimal policy (FLACK), it leverages whole-execution profiling and transient behavior detection to improve replacement decisions, thereby reducing micro-op cache misses and improving performance-per-watt in datacenter workloads. Ajorpaz et al. [22] use global history reuse prediction to eliminate dead entries both at the BTB and L1I. These techniques are orthogonal to ICARUS and can further enhance front-end performance.

3.4 Recent criticality and context-based microarchitecture optimizations

Load criticality-based data prefetching [32] highlights that conditional branches and branch history affect the loads and their criticality, resulting in dynamic-criticality behavior. Critical slice prefetching [33] performs data prefetching by identifying critical branch slices and tries to reduce branch misprediction penalties. The criticality-driven fetch [34] provides an execution paradigm that fetches, allocates, and executes instructions on the program's critical path. PHAST [35], a memory dependence predictor, uses the information about the path taken from the store to its dependent load and the store distance. ICARUS is orthogonal to these proposals as none target improving L2 instruction caching.

Chapter 4

ICARUS L2 Cache Replacement Policy

The analysis presented in the introduction (Figure 1.2) shows an average speedup of 19.80% when all instruction accesses hit in L2. The state-of-the-art L2 replacement policy EMISSARY advocates for a front-end criticality-based replacement policy for instructions in L2, which is a promising direction. Although EMISSARY improves performance, we observe some missed opportunities. Table 1.1 shows that a large fraction of instruction fetches do not cause decode starvation and an empty issue queue and are not critical. On average, only 3.49% of the instruction fetches are critical. According to Table 1.1, these 3.49% of the instruction fetches cause an average of 23.18% front-end stalls due to decode starvation. Ideally, a replacement policy should provide L2 hits to these critical instruction fetches. However, identifying critical instruction fetches (and critical lines) is not straightforward, as we observe the dynamic nature of criticality behavior (a particular instruction line sometimes behaves as a critical line and sometimes does not).

4.1 Identifying the dynamic behavior of critical fetches

In this Section, we argue for new ways to detect critical instruction fetches that show dynamic behavior in terms of criticality (d-critical). Figure 4.1 shows the percentage of s-critical and d-critical instruction lines (based on all fetched instructions that have been critical at least once). An instruction line that behaves critically for more than 95% of the instruction fetches is termed s-critical. The figure shows that 71.68% of the critical instruction lines are d-critical. A small fraction (28.32%) of instruction lines persist in their criticality behavior,

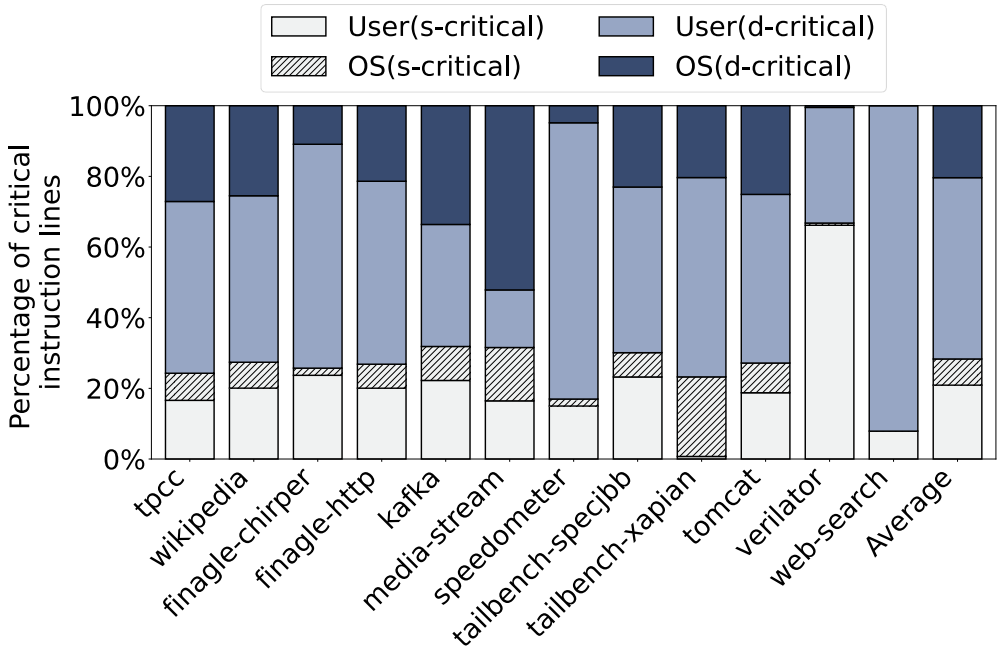


Figure 4.1: Distribution of critical instruction lines (s-critical and d-critical)

Table 4.1: Effect of context (branch history) on decode starvation behavior for `finagle-chirper`

PC	Branch history	Decode starvation?
0xFFFFE8C8A240	**1**1000100	Always
	01000100	Never

which means that only 28.32% of the lines cause decode starvation the majority of times they are fetched, and a significant fraction of critical lines may or may not cause decode starvation in their future fetches. Interestingly, the dynamic behavior is visible across OS and user-level instruction lines.

4.2 An additional context(branch history) for critical fetch detection

We observe that the control flow has a significant impact on the criticality behavior of the instruction fetch. One way to capture control flow is to examine the branch history. We look at the last k predicted branches and encode the history into a string of 1s (taken) and 0s (not taken). Table 4.1 shows an example for the benchmark `finagle-chirper` for a

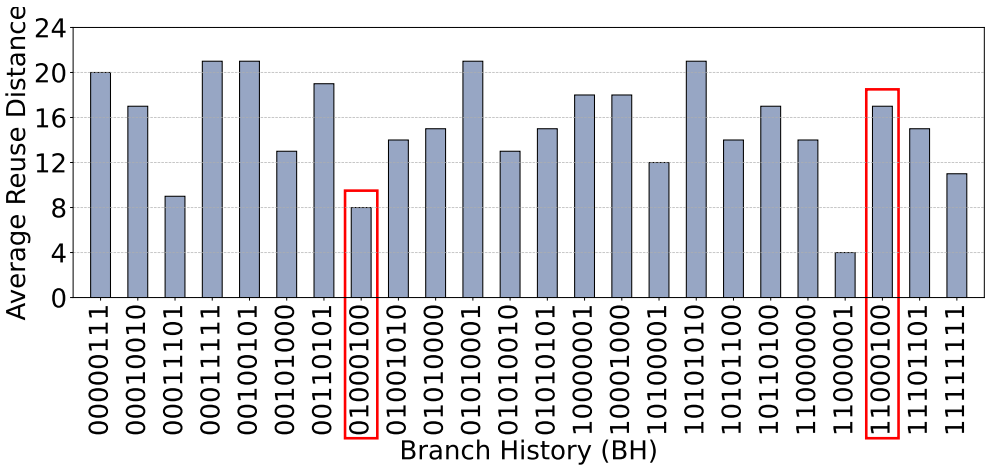


Figure 4.2: Variations in local reuse distance(set-based) at 16-way associative L2 with different branch histories for PC `0xFFFFE8C8A240` of `finagle-chirper`. Branch histories used in Table 4.1 are highlighted.

cache line aligned program counter (PC) `0xFFFFE8C8A240` with two different branch histories (rightmost bit denotes the recent branch prediction) and its corresponding decode starvation behavior due to an L2 miss. As we can see, the exact cache line address causes decode starvation if the branch history is "11000100"; however, for the history of "01000100", it never leads to decode starvation. This motivates the inclusion of branch history in identifying critical instruction cache lines.

4.2.1 Why branch history affects criticality behavior?

We observe that an instruction fetch shows different criticality behavior based on different branch histories as the reuse distance of an instruction cache line at L2 varies based on the branch history (Figure 4.2). Specifically, longer local reuse distances (greater than 16) tend to result in an L2 miss, leading to decode starvation. In contrast, branch histories associated with shorter local reuse distances result in L2 hits, thereby reducing the likelihood of decode starvation. Combining Table 4.1 and Figure 4.2, we can see that for PC `0xFFFFE8C8A240`, branch histories **1**1000100 and **0**1000100 have local reuse distances of 17 (L2 miss) and 8 (L2 hit), respectively. Figure 4.3 illustrates this behavior using a simplified control flow diagram for `finagle-chirper`. Depending on the branch outcome, execution flows through either block one (Ⓐ) or block two (Ⓑ), both of which lead to a common function `func`. When execution follows Block 1, `func` is invoked after approximately 10^6 instructions. However,

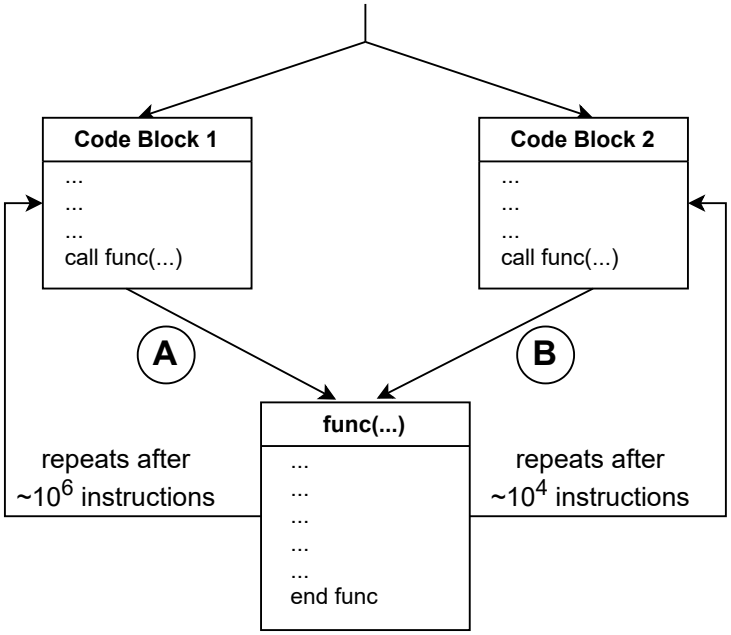


Figure 4.3: Control flow diagram of `finagle-chirper`.

when the execution flows through Block 2, `func` is invoked after every 10^4 instructions. This variation in control flow results in differences in branch histories, leading to significant differences in reuse distance for the same PC, thereby contributing to different criticality behavior. Please note that the branch history is affected by both user-level and OS-level instructions, as well as the interdependencies within the system stack.

4.2.2 Effect of branch history on decode starvation

Next, we quantify the effect of branch history and analyze the decode starvation behavior based on PCs and their branch history in Figure 4.4, where the first bar represents PCs and the second represents a tuple of PC and its branch history segregated based on the percentage of times that decode starvation is caused across their accesses. So, each benchmark in the X-axis has three signatures: PC and two different hash functions that are used to detect the criticality of an instruction line. There are three bins in the bar graph: 0%-5%, 5%-95%, and 95%-100%. If a PC almost always causes decode starvation, it appears in the bin 95%-100%. Similarly, if a PC rarely causes decode starvation, it appears in the 0%-5% bin. The PCs that exhibit dynamic behavior regarding decode starvation are located in the middle bin:

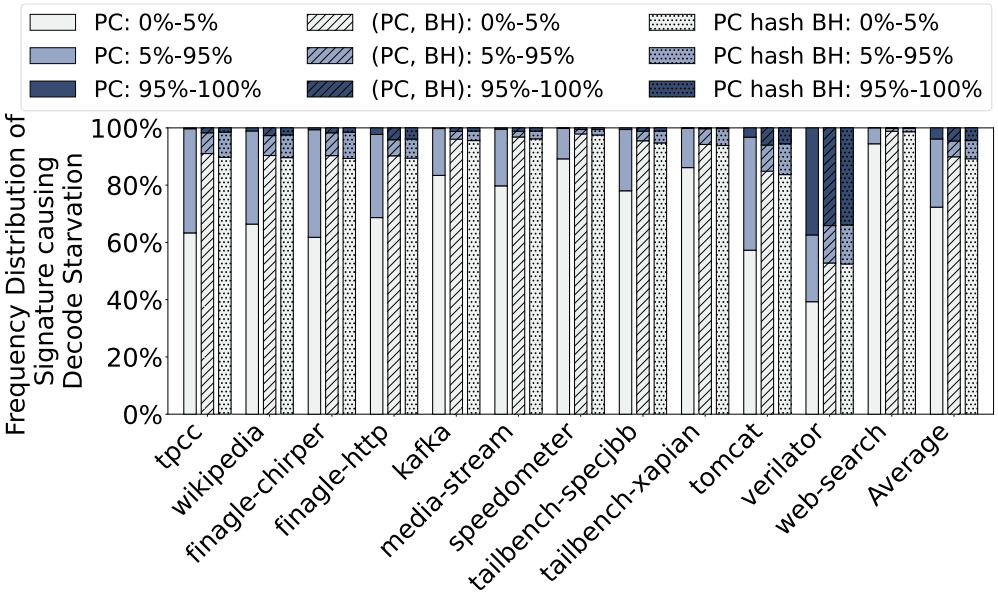


Figure 4.4: Distribution of PCs, tuple of PC and branch history (BH) of size nine, and PC hashed with BH categorized based on the percentage of occurrences causing decode starvation

5%-95%. Ideally, a detection mechanism that detects critical fetches with high accuracy should have fewer PCs in the 5%-95% bin and more PCs in the two extreme bins.

Figure 4.4 shows the effect of using just the PC as context information (signature) to identify critical fetches (similar to EMISSARY). Next, we show the impact of combining PC with branch history as a signature to detect whether a particular instance of an instruction fetch with a given branch history will lead to decode starvation. This is the ideal signature (hash function), though it is challenging to implement due to storage overhead. Finally, we use a hash of PC and branch history and show that we are closer to the ideal detector. The combination of PC with branch history is a well-known hash from initial branch predictor designs like GShare [36]. We observe that, on average, with PC as the signature, only 3.87% of PCs cause decode starvation more than 95% of the time, and almost 24% of PCs cause decode starvation between 5% and 95% of the time. This distribution becomes more concentrated if we take branch history into account with the PCs, as 4.6% of PCs cause decode starvation more than 95% of the time, while between 5%-95%, there is a frequency of only 5.5% (a drop from 24%). From this figure, it is evident that the decode starvation behavior of an instruction line significantly depends on the branch history that precedes the fetch of that instruction line.

4.3 Critical fetch detection

To detect critical instruction fetches, we utilize a critical instruction fetch identification table (CIT), also called a criticality indicator table, with n entries, each using a k-bit saturating counter. The cache line address of every fetch that sees a decode starvation is hashed with the branch history. We index the CIT with the hash, and for that particular index, we increment the counter every time we see a critical instruction fetch coming from L2, L3, or DRAM (❶ ❷ ❸ of Figure 4.5). If the counter value is greater than a threshold (two), then we send a criticality signal to the L2, marking that L2 line as critical (❹❺ of Figure 4.5).

Please note that with this threshold of two on the counter, we filter out dead L2 lines that are critical. We use the term "dead" for lines with no reuse (dead-on-arrival). The threshold of two ensures that a line causes decode starvation at least twice before it is marked as critical. Thus, dead-on-arrival decode-starvation-causing lines are never marked as critical. As we filter them, all critical lines see at least one reuse.

Next, because we do not decrement the counters in the critical fetch identification table, we may end up over-predicting critical lines. To mitigate that, we flush the table once in x million cycles and restart the learning from scratch. For our implementation, we use a table of 512 entries, a branch history of nine bits, and a saturating counter of two bits with a threshold of two, and we flush the criticality table once in one million cycles. We do not use any replacement policy as CIT is tagless, which allows aliasing. We use the following hash function to index the CIT.

$$line\_addr = pc \gg \log_2 (CACHE\_LINE\_SIZE) \tag{4.1}$$

$$\begin{aligned} \text{signature} = (line\_addr \oplus (line\_addr \gg 3) \oplus bh) \\ \&\ \ (\text{CIT\_SIZE} - 1) \end{aligned} \tag{4.2}$$

4.3.1 Effect of BHC on decode starvation

Figure 4.6 shows the effect of using PC with additional context information in the form of branch history (BHC) for identifying critical fetch with the underlying L2 replacement policy as EMISSARY. Compared to TPLRU, PC with branch-history-based EMISSARY reduces decode starvation cycles by 6.5%, whereas EMISSARY without context reduces them by 2.5%. Compared to the baseline TPLRU policy, if we exclude `Verilator`, then EMISSARY with the context information reduces decode starvation cycles by 7.1%, which was 1% with EMISSARY without the context.

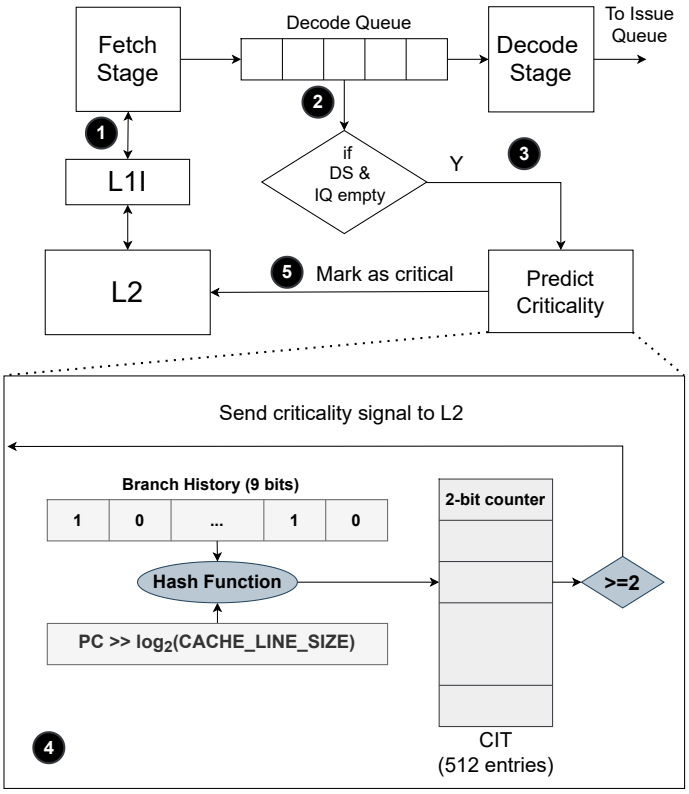


Figure 4.5: Critical instruction fetch detection with CIT. DS: Decode starvation and IQ empty: Issue queue empty.

4.4 Criticality, but also reuse

Figure 4.7 shows the speedup of BHC and EMISSARY over the baseline TPLRU with FDIP. BHC achieves performance comparable to, and in some cases slightly better than, the state-of-the-art EMISSARY across most benchmarks. However, the improvements are often negligible, and there are still some exceptions where this trend does not hold. This raises an important question: Is criticality alone sufficient for designing an L2 cache replacement policy that preserves cache lines based on the instruction criticality? Cache replacement policies are primarily driven by reuse. Therefore, a front-end criticality-aware replacement policy must also account for the reuse behavior of both critical and non-critical cache lines.

As per Figure 4.8, once a line becomes critical, it shows that it will get reused (irrespective of the local reuse distance) as we filter out dead-on-arrival critical lines. Please note that the local reuse distance of a line refers to the reuse distance calculated within its cache set.

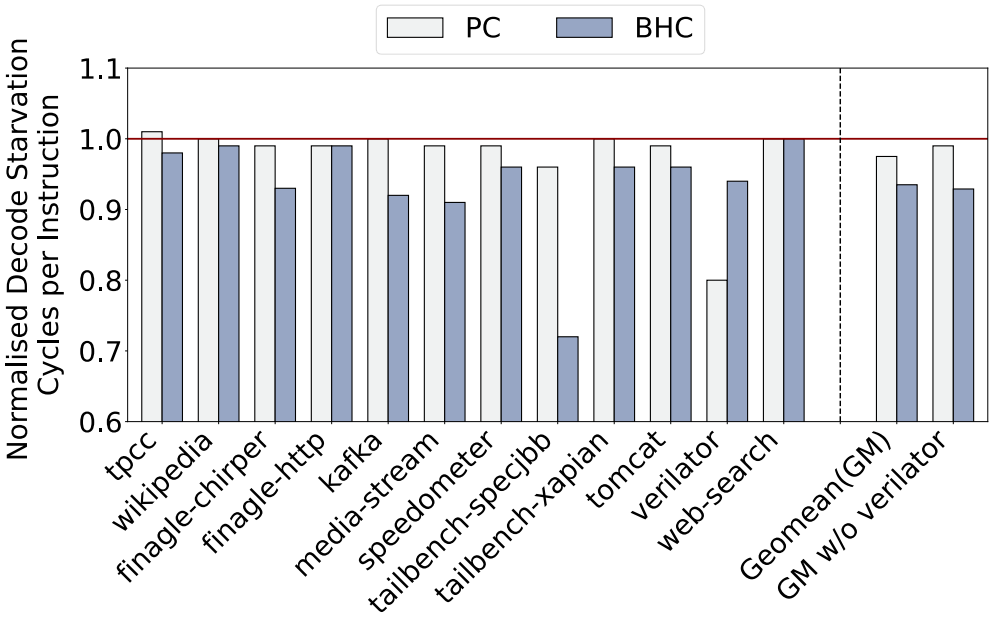


Figure 4.6: Decode starvation cycles per instruction normalized to TPLRU, for EMISSARY with PC-based signature and PC with branch history(BHC) based signature

Please refer to C:Short, C:Mid, C:Long in Figure 4.8. Also, with our context-based hash and the threshold used for the counter, our indexing in the criticality indicator table filters out dead critical lines (C:No reuse is nil in Figure 4.8). On the other hand, around 19% of non-critical lines are *dead*, and they get no reuse. So, EMISSARY rightly decides to isolate critical lines from non-critical lines so that non-critical lines cannot evict critical lines that will get reused. EMISSARY evicts non-critical lines unless the critical lines reach a certain threshold (N). If the number of critical lines is greater than N, then it evicts critical lines.

As per Figure 4.8, on average, 19.10% of the critical lines have a short reuse distance of less than 16 (C:Short in Figure 4.8), and as EMISSARY isolates critical lines from non-critical ones, on average, EMISSARY provides hits for most of the critical lines with short reuse distance even with reduced associativity of eight. However, due to its limited associativity, EMISSARY fails to provide additional cache hits for some mid-reuse critical lines. The same trend continues for long reuse cache lines with reuse distances greater than 32. Please note that the number of critical lines with mid and long reuse distances is extremely small compared to the number of non-critical lines, corroborating our observation in Chapter 1, which shows that only 3.49% of instruction fetches lead to decode starvation. Finally, there are a few non-critical lines with mid and long reuse distances that, if kept in the cache for a longer duration, will start providing L2 hits. However, the utility of these L2 hits is marginal

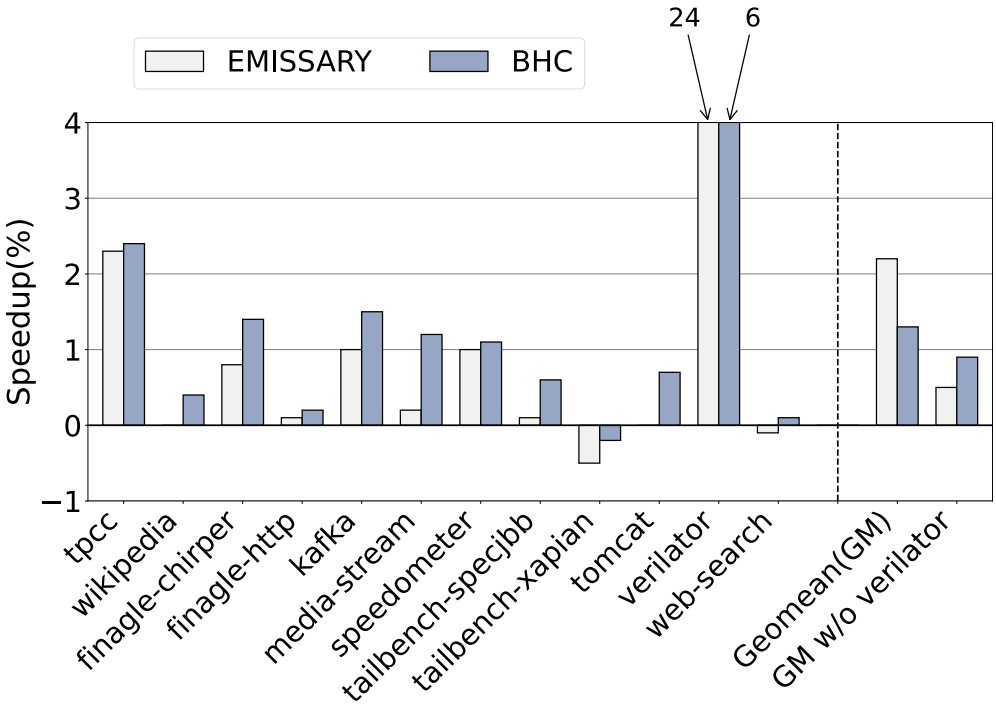


Figure 4.7: Speedup of BHC relative to TPLRU

as they do not cause decode starvation. Please note that a non-critical cache line can become critical the next time it comes to the L2, thanks to the dynamic behavior of criticality.

In summary, based on our definition of criticality, a critical cache line is always reused, regardless of the local reuse distance. So, a critical line that has yet to be reused should be preserved at the L2. EMISSARY does not consider the effect of reuse and treat all critical lines, equally. As a result, it misses the opportunity to provide L2 hits to mid- and long-reuse critical lines.

4.5 BRC: Criticality and Reuse-based Bins for L2 replacement policy

We segregate L2 lines with a local reuse distance of 0-16 from cache lines having a local reuse distance of [16-32). We keep the lines with a local reuse distance of [16-32) a bit longer in the cache so they get a hit on subsequent accesses. We achieve this segregation using a straightforward method. We associate each L2 line with an additional bit, the *reuse* bit. Initially, the bit is set to 0 when the cache line is first filled. If the cache line gets a hit in its

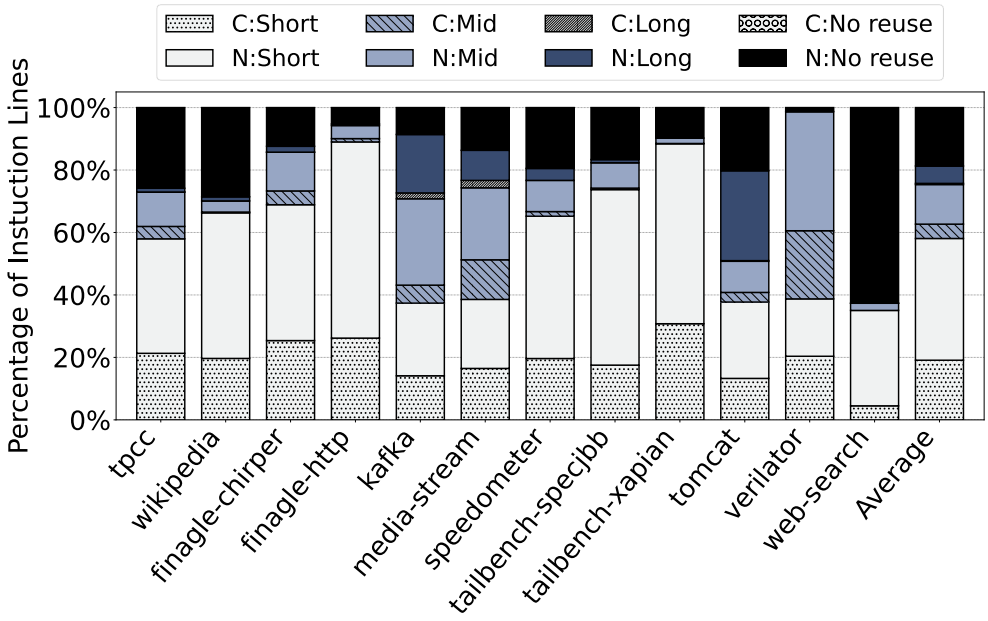


Figure 4.8: Local reuse distance of instruction lines at L2: short [0-16), mid [16-32), long 32+, and no reuse for both critical (C) and non-critical (N) lines. We use the hash of PC with branch history for detecting critical instruction fetches. Note that this study is independent of an L2 replacement policy.

lifetime, we set the bit to one. In short, we classify cache lines of an L2 set into four bins: critical and not reused [1, 0]; critical and reused [1, 1]; non-critical and reused [0, 1]; and non-critical and not reused [0, 0] as seen in Figure 4.9. Based on the reuse bit, we aim to optimize the eviction policy to increase hits on mid-reuse and long-reuse L2 lines. We name these bins Criticality and Reuse-based Bins (BRC), where eviction decisions are made based on two bits (criticality and reuse) per L2 line.

4.6 ICARUS: Incorporating BHC and BRC

ICARUS stands for Instruction Criticality and Reuse-based L2 cache replacement policy. ICARUS (BHC+BRC) utilizes branch history to identify critical fetches and employs a bin-based replacement policy based on criticality and reuse.

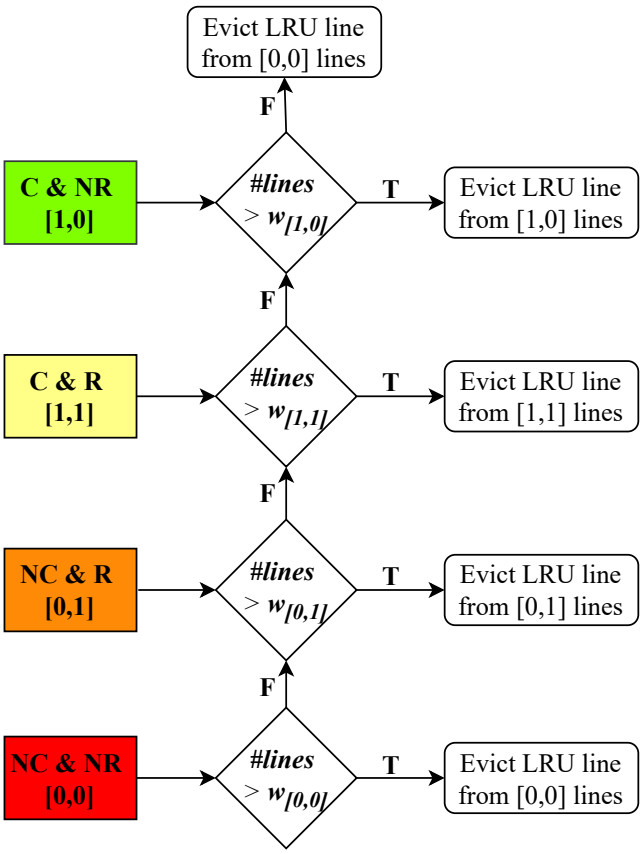


Figure 4.9: BRC eviction policy. The L2 lines are segregated into four bins: C&NR, C&R, NC&R, NC&NR, where each bin has its watermark (w[x,y]). The search for an eviction candidate follows the priority order from red to green based on the respective watermarks.

4.6.1 Eviction policy

To allow critical instruction lines with high local reuse distance (32+) to remain long enough in the cache, we assign the highest priority to the cache lines that are critical and not reused (critical bit is one, but reuse bit is zero) to preserve them in L2. As per Figure 4.8, all critical lines get a reuse, which means critical lines with the reuse bit zero will get a reuse in the future. However, this is not true for non-critical lines. Next, we try to keep critical lines that are reused (the critical bit is one, and the reuse bit is one). Next, we prioritize non-critical lines that are reused (the critical bit is zero and the reuse bit is one) and, finally, non-critical lines that are yet to be reused (both the critical and reuse bits are zero).

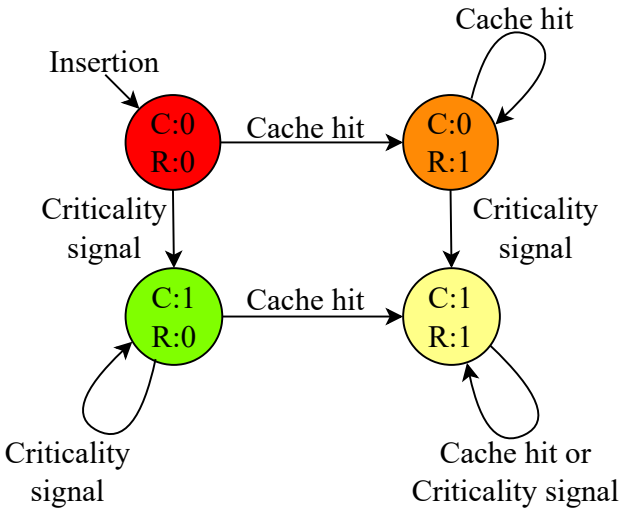


Figure 4.10: State transitions of L2 lines with ICARUS. C denotes the criticality bit, and R denotes the reuse bit.

Based on the bins at L2 as discussed in the above section, the ICARUS eviction policy decides which eviction bin to use, then which cache line to select from that bin. Please note that we use the Tree-based Pseudo LRU (TPLRU) bit, in addition to the critical and reuse bits. As the L2 is shared between data and instruction lines, additional protection for instruction lines at the expense of data lines will lead to a performance drop. So, we use TPLRU bits that maintain the TPLRU chain across all lines within a set, regardless of whether they contain data or instructions. All data lines are assigned to two bins: non-critical and reused, and non-critical and yet to be reused, along with their respective PLRU bits, resulting in a 2-bit RRIP policy across data lines. If we remove the *reuse* bit from our policy, it will behave the same as EMISSARY, but with BHC.

Figure 4.9 shows the eviction policy that evicts lines from low priority [0,0]. Overall, the eviction sequence among four bins is $[0,0]$, $[0,1]$, $[1,1]$, and $[1,0]$. We use watermarks for each bin that set the minimum threshold for each bin before we start evicting lines from that bin. Based on the criticality and reuse, we use the watermarks of two, four, six, and four for four bins: [0,0],[0,1],[1,1], and [1,0]. Note that bin [1,1] needs more space than [1,0] because most of the [1,0] lines will eventually become [1,1], and on top of that, [0,1] lines can also become [1,1]. In case of a tie, i.e., each bin has occupancy equal to its watermark, our eviction policy evicts lines from the low-priority bin. However, on average, less than 0.10%

of L2 evictions lead to a tie. With BRC, more than 99.6% of L2 evictions are non-critical instruction lines. With these watermarks, we ensure an appropriate L2 space for each bin without causing misses for other bins.

4.6.2 Insertion and promotion policy

Figure 4.10 shows the promotion policy with BRC. An L2 line is inserted in the MRU position with the criticality bit set to zero and the reuse bit set to zero. On a reuse or a criticality signal from the CIT, the cache line moves from one bin to another.

4.7 How to choose watermarks?

For the 12 applications that we studied, watermarks of two, four, six, and four for bins [0,0],[0,1], [1,1], and [1,0] perform the best. We perform extensive experiments and sweep through all possible values for these watermarks. Based on our experiments, we conclude the following: (i) for bin [0,0], if we use more than two as the watermark, it starts occupying L2 space, affecting critical instruction lines. For applications like `Verilator`, for bin [0,0], a watermark of zero performs the best as the code footprint is huge and 19.89% of the instruction fetches are critical. So, decreasing the watermark for bin [0,0] ensures the eviction of non-critical and non-reused lines immediately. However, this is not the case with other applications. (ii) for bin [0,1], the watermark of four provides enough time to ensure a transition (if there is a possibility of a transition) from [0,1] to [1,1] as 1.2% of all transitions are from [0,1] to [1,1]. Increasing this watermark does not improve performance further. However, if we decrease it, applications like `finagle-chirper` and `tomcat` get affected. On average, less than 6.32% of transitions are from non-critical L2 lines to critical L2 lines. (iii) For bins [1,1] and [1,0], watermarks of six and four provide sufficient time for providing hits to long-reuse critical lines. We keep [1,1] lines for more time compared to [1,0] as we observe that once a line goes to [1,1], it gets at least one more reuse. When we increase these watermarks further, there is no utility. Decreasing these watermarks further degrades performance marginally. However, if some of the watermarks are less than four (e.g., 4-6-3-3 and 4-8-2-2), especially for both [1,1] and [1,0], performance drops by more than 2%.

Chapter 5

Evaluation

5.1 Simulation methodology

We use the gem5, a widely used cycle-accurate simulator with FDIP [19, 37], to run a detailed CPU model in Full System (FS) mode with Ubuntu 18.04 (Linux kernel version 4.15). We use the simulation infrastructure developed by the authors of EMISSARY and the checkpoints shared by the authors of EMISSARY as part of the artifact [38]. We use 12 datacenter applications for our evaluation, as detailed in Section 5.1.1 below. Table 5.1 details the simulated processor and memory hierarchy. Our simulation parameters are similar to Intel's Granite Rapids [15] cache hierarchy, configured with the TPLRU baseline as the L2 replacement policy. We use the existing TPLRU bits without adding new state or bits. Insertion and promotion of blocks at L2 follow the standard TPLRU behavior; only the eviction decision is modified based on the ICARUS eviction policy. We use the FDIP-based instruction prefetcher [16] in the baseline, and the cache hierarchy is non-inclusive. We simulate each datacenter benchmark for 200 million instructions after a warm-up of 50 million instructions. Table 1.1 characterizes the evaluated benchmarks based on instruction footprint, branch predictor accuracy, critical instruction fetches, CPU stalls caused by critical fetches, and cache misses per kilo instructions (MPKI), separated by instructions and data.

5.1.1 Description of benchmarks

We evaluate our approach using a diverse set of benchmarks that represent a wide range of real-world workloads. From the OLTP-Bench suite, we include *tpcc*, an online transaction processing application, and *wikipedia*, a MediaWiki-based workload operating on a Wikipedia

Table 5.1: Simulated parameters

Granite Rapids-like CPU	
Fetch Target Queue	24 entry 192-instructions, FDIP [16]
Branch Predictor	TAGE [21], ITTAGE [39]
BTB	16K entries
Fetch/Decode/ Commit width	6 / 6 / 8
ROB	512 entries
Issue/Load/Store Queue	240 / 192 / 114 entries
Int/FP Registers	280 / 332
Granite Rapids-like cache hierarchy	
Private L1I/L1D	64KB(I), 48KB(D), 8/12 way, 64B line, TPLRU, 6 and 5 cycles, 16 MSHRs
Private unified L2	2MB, 16-way, 64B line size, TPLRU, 16 cycles, 32 MSHRs
Shared L3	3MB/core, 12-way, 64B line, TPLRU, 33 cycles, 64 MSHRs, non-inclusive

dataset. From the Renaissance suite, we use *finagle-chirper*, a microblogging service by Twitter, and *finagle-http*, which models a production-grade HTTP server.

We also include workloads from the DaCapo benchmark suite, such as *kafka*, a distributed event streaming platform, and *tomcat*, an implementation of Jakarta Servlet, Expression Language, and WebSocket technologies. From CloudSuite V4, we use *media-stream*, which simulates video streaming traffic, and *web-search*, based on the Apache Solr search engine.

Additionally, we include *speedometer*, a browser-based JavaScript benchmark that measures web application responsiveness, and *verilator*, which simulates the RTL design of Rocket Chip while executing a quicksort workload. From the Tailbench suite, we incorporate *specjbb*, a SPEC benchmark for evaluating Java application performance, and *xapian*, a web-search application.

Together, these benchmarks provide comprehensive coverage of server-side, cloud, and interactive workloads, enabling a thorough evaluation of the proposed approach.

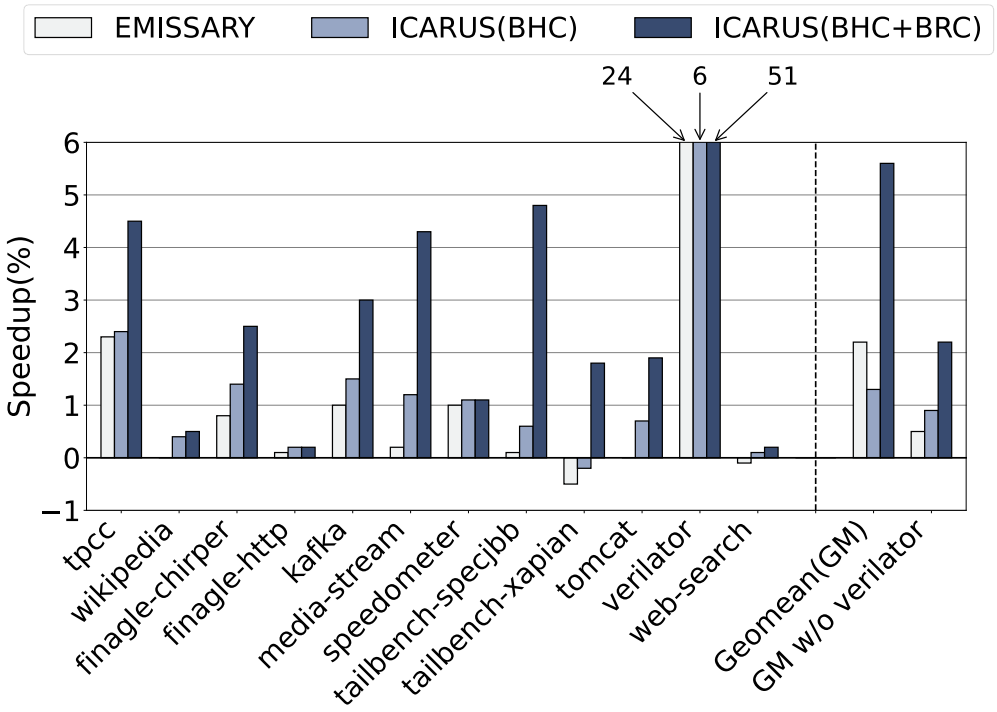


Figure 5.1: Speedup relative to TPLRU

5.2 Performance evaluation

5.2.1 Performance

We use improvement in runtime as the metric for performance. As a baseline L2 replacement policy, we use the TPLRU policy. We compare ICARUS with EMISSARY, as EMISSARY is the state-of-the-art replacement policy. We use the authors' version of EMISSARY, available in the public domain [37]. Figure 5.1 shows the performance improvement across all applications with EMISSARY, ICARUS having branch history criticality (BHC), and ICARUS with both branch history criticality and criticality and reuse bin-based eviction policy (BHC+BRC). On average, ICARUS (BHC+BRC), our final contribution, improves performance by 5.6% as high as 51% for `Verilator` while the performance excluding `Verilator` is 2.2%. EMISSARY, on the other hand, delivers a performance improvement of 2.2%, which becomes 1% without `Verilator`. For `tailbench-xapian`, there is a marginal performance drop (less than 0.2%) with ICARUS (BHC) as the data misses increase by 5.06%. However, with ICARUS (BHC+BRC), we mitigate this performance drop and improve performance by 1.8%.

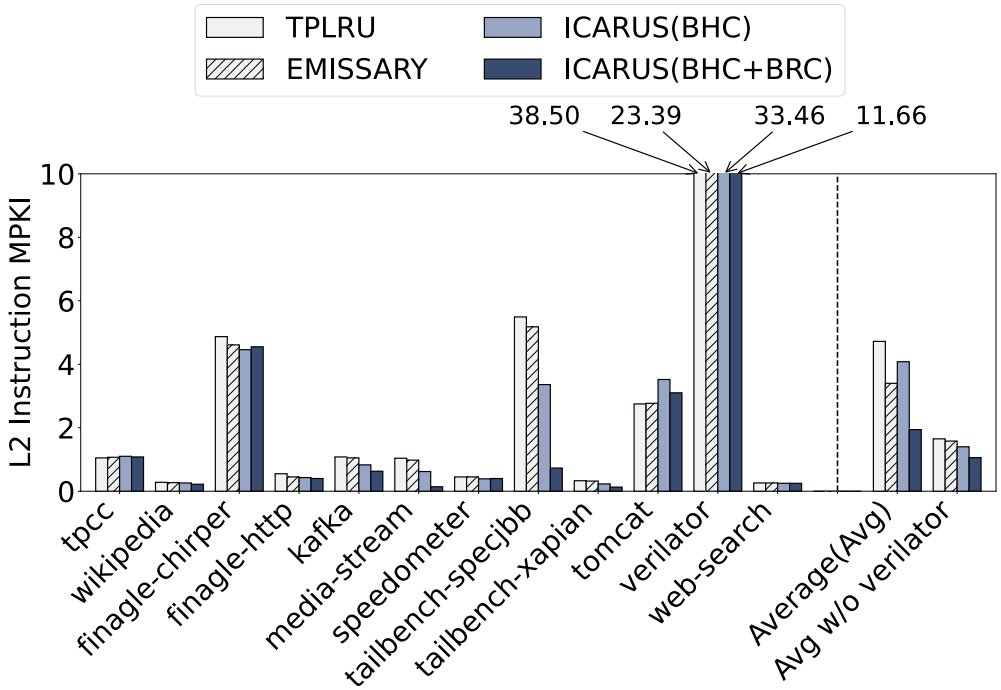


Figure 5.2: L2 instruction MPKI

5.2.2 Instruction MPKI and decode starvation

Figure 5.2 shows the reduction in instruction MPKI at the L2. On average, ICARUS reduces instruction MPKI from 4.72 to 1.94. The reduction in MPKI results in a reduction in decode starvation cycles (because of an L2 miss) per instruction. Figure 5.3 shows decode starvation cycles per instruction normalized to TPLRU. ICARUS outperforms EMISSARY in all benchmarks, reducing the front-end bottleneck because of an L2 miss, significantly (an average reduction in decode starvation cycle per instruction from 0.97 to 0.86). ICARUS improves the instruction MPKI but not at the cost of a significant increase in data MPKI. We observe an interesting trend for `tomcat`, where there is an increase in instruction MPKI with ICARUS (Figure 5.2). However, as per Figure 5.3, there is a reduction in decode starvation cycles per instruction. So, the increase in MPKI is actually for the non-critical lines, and because of this, an increase in MPKI does not affect performance. This trend is also a side effect of our replacement policy, prioritizing critical lines over non-critical lines.

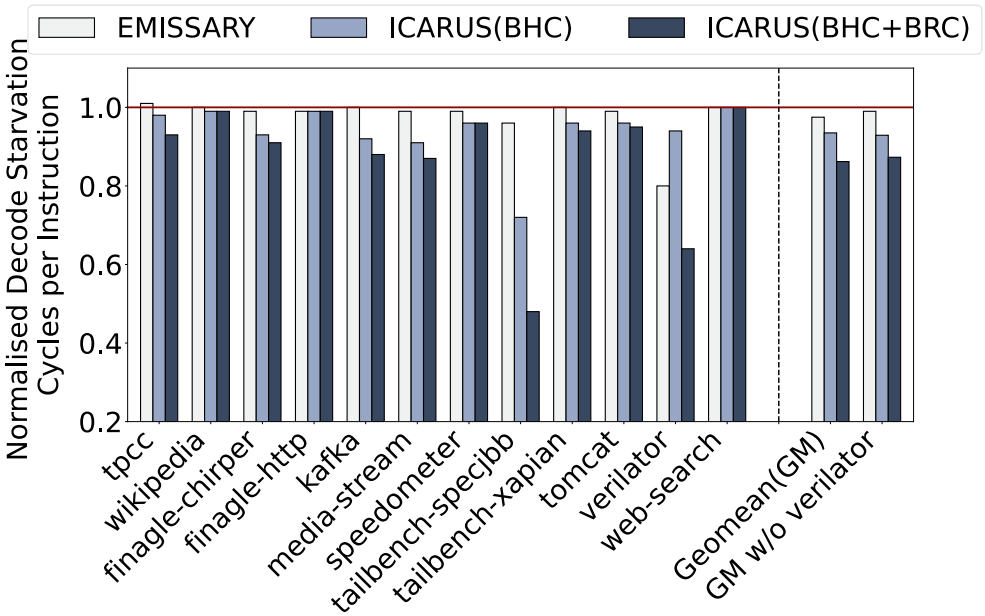


Figure 5.3: Decode starvation cycles per instruction normalized to TPLRU

5.2.3 Instruction MPKI based on their reuse

One of the key contributions of ICARUS (BHC+BRC) is to keep the critical instruction lines that will be reused (mid and long local reuse distance). Figure 5.4 shows the decrease in misses per kilo instructions (MPKI) of instruction lines at the L2. As per Figure 5.4, ICARUS (BHC+BRC) reduces the MPKI of short reuse lines marginally as compared to ICARUS (BHC). However, the primary difference comes from the ICARUS (BHC+BRC) when it comes to mid reuse lines, as it significantly reduces the MPKI of instruction lines that are critical and have high reuse distance. For long reuse instruction lines, the figure shows the average trend, where the MPKI drop appears marginal, which is 0.1 to 0.09. However, certain workloads, such as `Kafka`, benefit significantly, where the MPKI of long reuse lines drops from 0.8 to 0.02 with ICARUS (BHC+BRC). Regarding the data lines at the L2, ICARUS (BHC+BRC) reduces the MPKI of both mid- and long-reuse data lines, with a slight increase in MPKI for short-reuse data lines. Note that criticality plays no role among data lines, and the effect in terms of MPKI is solely based on reuse (Figure 5.5).

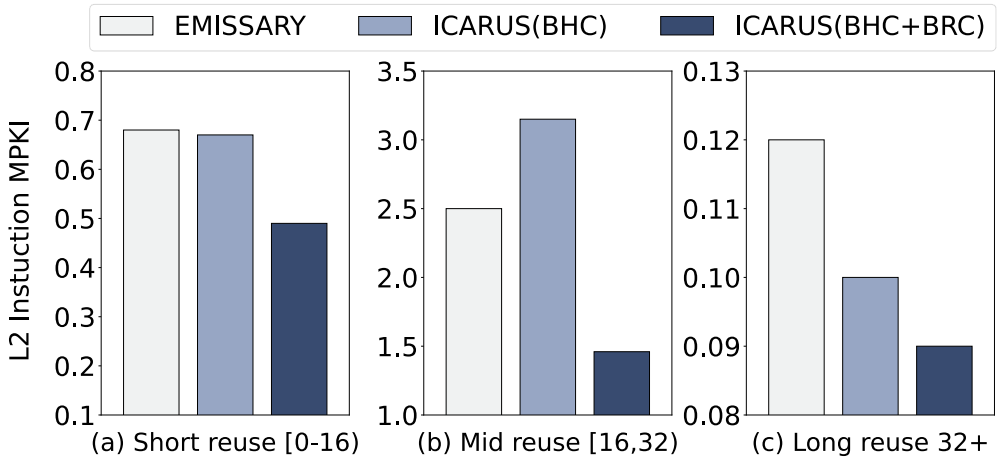


Figure 5.4: L2 instruction MPKI segregated based on reuse

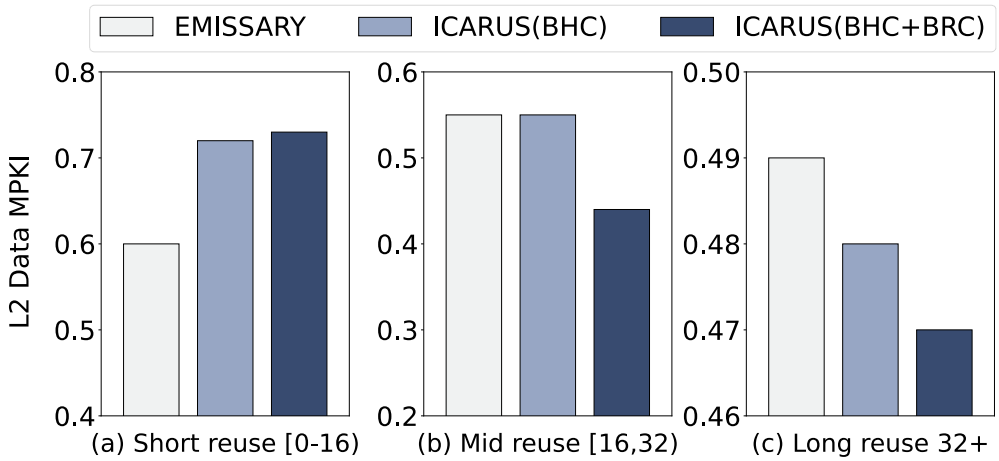


Figure 5.5: L2 data MPKI segregated based on reuse

5.3 Interaction with instruction and data prefetchers

5.3.1 Interaction with instruction prefetcher

Priority Directed Instruction Prefetching (PDIP) [6] is a recent instruction prefetching technique that provides a good performance boost on top of the EMISSARY replacement policy. PDIP is a front-end criticality-based instruction prefetcher that considers instruction fetches missed in the L1I and caused decode starvation. We quantify the effect of the best version of ICARUS in the presence of PDIP as an instruction prefetcher on top of FDIP. Figure 5.6 shows the effect of PDIP without EMISSARY and ICARUS (best combination of ICARUS among all proposed techniques), where PDIP alone improves performance by 2.5% with

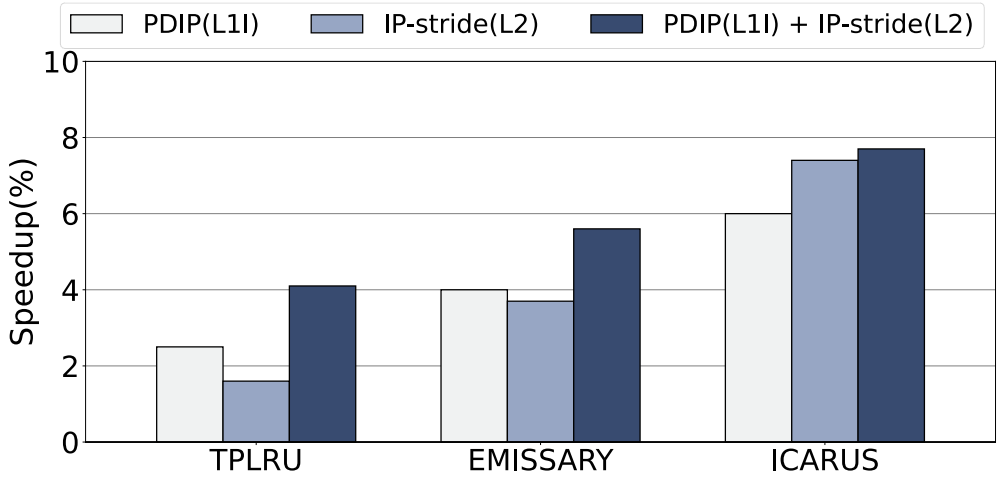


Figure 5.6: Speedup with different prefetching techniques normalized to no data prefetching and FDIP instruction prefetching.

TPLRU, and the combination of PDIP+EMISSARY improves performance by 4% compared to the baseline with TPLRU as the replacement policy and FDIP as the instruction prefetcher. PDIP with ICARUS improves performance by 6%. The improvement comes from additional L1I hits because of PDIP. PDIP and ICARUS work synergistically as ICARUS keeps critical lines longer, and PDIP prefetches these lines into L1I. As some of the critical instruction lines from L2 are also causing decode starvation even on an L2 hit, L1 hits to those instruction fetches mitigate decode starvation cycles further.

Interaction with data prefetcher Next, we quantify the effect of the IP-stride data prefetcher at L2 that is commercially implemented in AMD and Intel processors. Figure 5.6 shows the effect of IP-stride without EMISSARY and ICARUS, where IP-stride at the L2 improves performance by 1.6% with TPLRU, and the combination of IP-stride+EMISSARY improves performance by 3.7% compared to the baseline with TPLRU as the replacement policy and FDIP as the instruction prefetcher. IP-stride with ICARUS improves performance by 7.4%. The improvement comes from additional L2 data hits. Next, we quantify the interaction of PDIP and IP-stride with EMISSARY and ICARUS. PDIP+IP-stride improves the average performance of EMISSARY by 5.6%, whereas with ICARUS, it goes up to 7.7%. Overall, ICARUS works synergistically with instruction and data prefetchers.

Table 5.2: Storage overhead with ICARUS

Structures	**Description**	**Storage**
CIT	512 entries, 2-bit saturating counter per entry	128B
Branch history register	For storing branch history	9 bits
Flags	Two one-bit flags for criticality and reuse for each cache line at the L2 of size 2 MB	8 KB
Total		**8.13 KB**

5.4 Storage Overhead

Table 5.2 summarises the storage overhead of ICARUS. For a 2MB L2 cache, the total storage requirement is 8.13KB. The dominant component of this overhead comes from the per cache line metadata, where two one-bit flags(criticality and reuse) are maintained for each L2 cache line, accounting for 8KB. The remaining structures, including the Critical Indicator Table(CIT) and the branch history register, contribute only a small fraction of the total storage. Compared to EMISSARY, which requires 4KB of storage, ICARUS incurs an additional overhead of 4.13KB. This increase is modest in the context of modern server processors, where cache sizes are in the order of megabytes. Despite this small increase in storage, ICARUS delivers significant performance improvements, making the tradeoff highly favorable.

The CIT is designed to efficiently track critical instruction fetches using a 512-entry table with two-bit saturating counters, resulting in a storage cost of only 128B. We empirically analyze the working set of instruction cache lines and observe that, on average, 272 unique instruction cache lines are accessed within a one million(1 M) cycle reset interval across all benchmarks, with a maximum of approximately 680 in workloads such as `tomcat`. This indicates that a 512-entry CIT is sufficient to capture the majority of critical instruction behavior. Increasing the table size beyond 512 entries provides negligible benefit (less than 0.01% improvement), while smaller sizes can lead to performance degradation due to insufficient coverage. Overall, ICARUS achieves a good balance between storage overhead and performance gain, demonstrating that a small amount of additional metadata can significantly enhance cache replacement decisions.

In addition to the metadata required for ICARUS, Tree-based Pseudo-LRU (TPLRU) bits are needed to identify the least recently used cache line within each bin. A TPLRU policy requires at most ((associativity - 1)) bits per set. Hence, for a 16-way associative cache, 15 bits are required per set. Since ICARUS maintains four bins, the total TPLRU

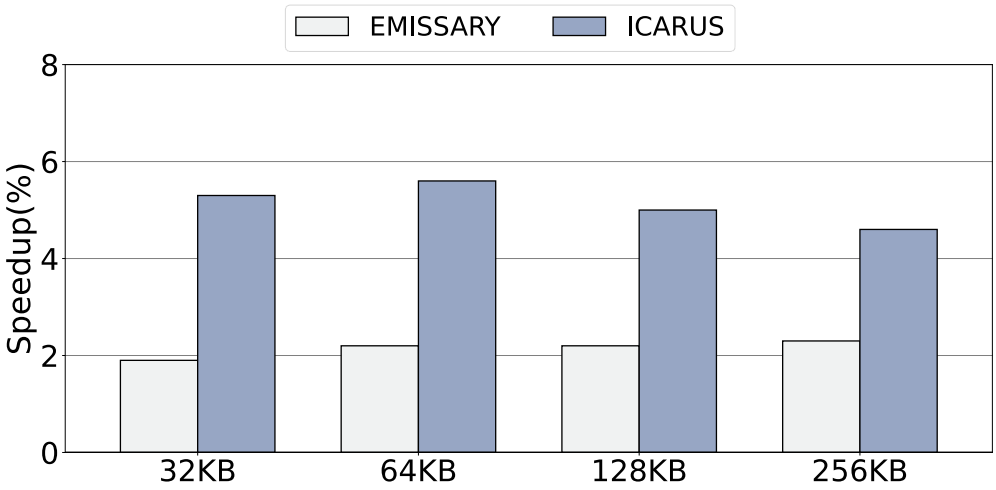


Figure 5.7: Sensitivity study: effect of L1I size on performance

overhead becomes 60 bits (7.5 bytes) per set. For a 2MB L2 cache, this approximates to 15KB of additional storage overhead for TPRLU. This overhead can be reduced to 11.5KB by allocating TPLRU bits based on each bin's effective occupancy rather than allocating 15 bits per bin.

5.5 Sensitivity studies

5.5.1 L1I size

Figure 5.7 shows the effect of L1I size on EMISSARY and ICARUS. There is a marginal decrease in speedup with the increase in L1I size. However, even for a large L1I like 256KB, ICARUS provides a significant speedup of 4.6% as compared to TPLRU, whereas EMISSARY provides a speedup of 2.2%. Overall, ICARUS outperforms EMISSARY across all L1I sizes. Server processors typically do not employ L1I caches larger than 256KB. Therefore, we limit our evaluation to this range and demonstrate that ICARUS remains effective even at larger L1I sizes.

5.5.2 L2 size

Figure 5.8 shows the effect of L2 size on EMISSARY and ICARUS. There is a gradual drop in performance gains as the L2 size increases from 1MB to 4MB. This trend is expected, as larger L2 caches can accommodate a large instruction working set, thereby reducing pressure

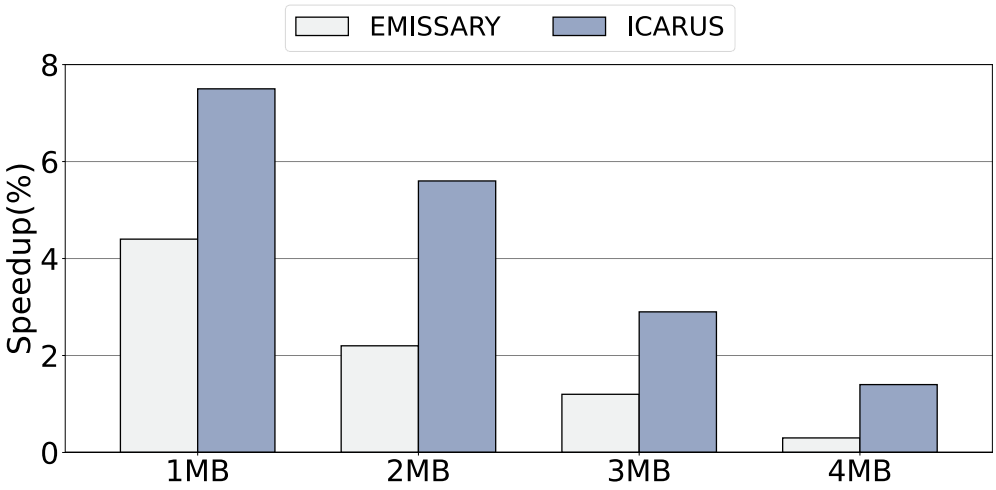


Figure 5.8: Sensitivity study: effect of L2 size on performance

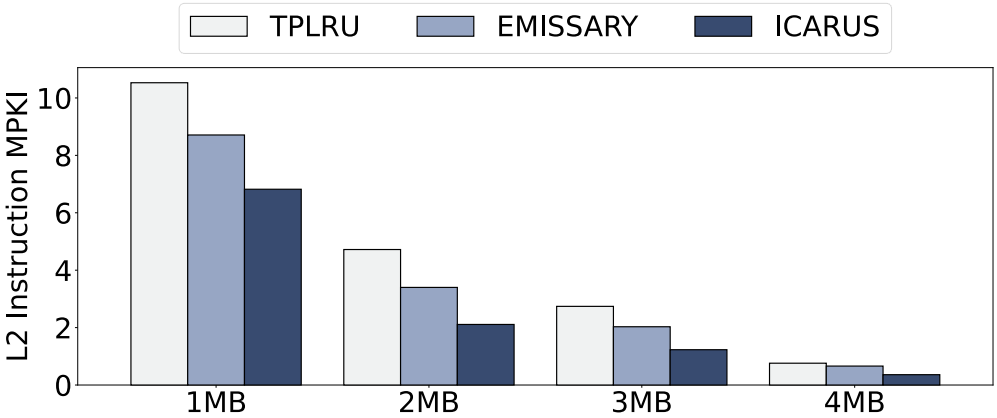


Figure 5.9: Average L2 instruction MPKI for different L2 sizes

on the replacement policy. Even for a large L2 of 4MB, ICARUS still achieves a measurable speedup of up to 1.4% compared to TPLRU, whereas EMISSARY provides only about 0.3% improvement. This behavior is primarily due to the fact that the code working set of most applications fits within a large L2 cache. As a result, the frequency of instruction cache misses at the L2 level is significantly reduced, leading to fewer opportunities for replacement policies. Consequently, the L2 instruction MPKI drops substantially, falling below 1 for a 4MB L2, as shown in Figure 5.9. Despite the reduced headroom at larger cache sizes, ICARUS consistently outperforms EMISSARY across all evaluated L2 configurations. This demonstrates that ICARUS is robust and continues to provide benefits even when cache capacity is less constrained, making it effective across a wide range of system configurations.

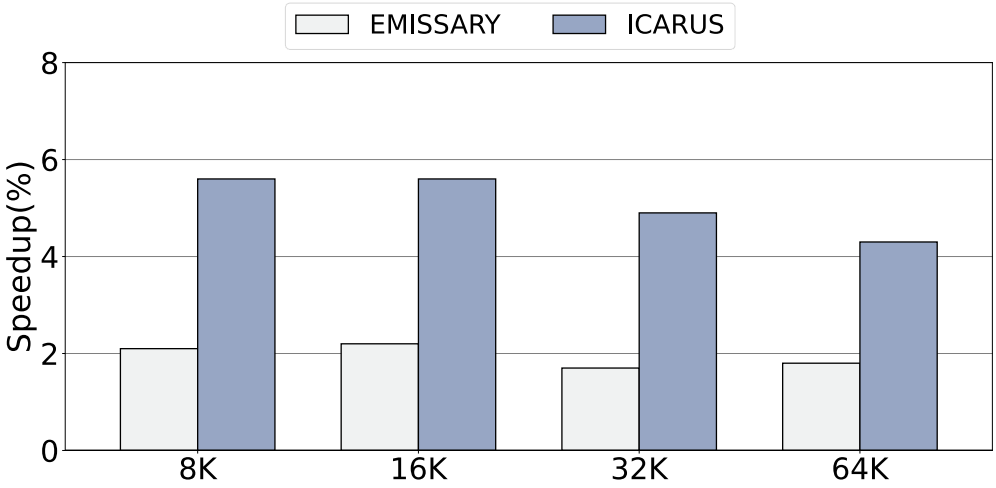


Figure 5.10: Sensitivity study: effect of BTB size on performance

It is important to note that the performance gain with ICARUS is lower for very large L2 sizes, such as 4MB, which may become more common in future server processors. However, this does not diminish the effectiveness of ICARUS. As discussed in Chapter 1, while cache sizes continue to grow, the code working set of applications is also increasing over time. This trend ensures that the role of ICARUS remains important for future datacenter processors.

5.5.3 BTB size

The Branch Target Buffer (BTB) is a critical structure in the processor front-end. In datacenter applications, which typically exhibit large instruction working sets, the BTB can become a performance bottleneck due to high pressure from numerous branch instructions. Therefore, it is important to understand how varying BTB sizes influence the effectiveness of ICARUS. Figure 5.10 shows the effect of BTB size on EMISSARY and ICARUS. As the BTB size increases, we observe a marginal reduction in performance gains for both EMISSARY and ICARUS. This trend is expected, since a larger BTB can accommodate more branch targets, thereby reducing miss rates and reducing front-end bottlenecks. Consequently, the opportunity for replacement policies to make impactful decisions diminishes with increasing BTB capacity. Despite this, ICARUS continues to deliver consistent improvements across all BTB sizes. Even for a large BTB with 64K entries, ICARUS achieves a notable speedup of 4.3% compared to TPLRU, whereas EMISSARY provides a speedup of only 1.8%. This demonstrates that ICARUS remains effective even when the BTB capacity is sufficiently large to capture most branch working sets. Overall, ICARUS outperforms EMISSARY

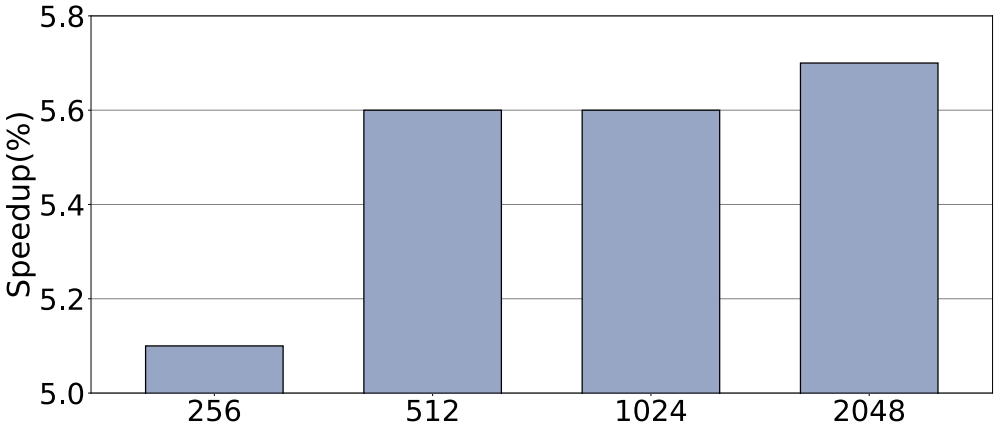


Figure 5.11: Speedup with ICARUS with different criticality indicator table(CIT) sizes normalized to TPLRU.

across all evaluated BTB sizes, highlighting its robustness and ability to improve front-end performance under varying BTB configurations.

5.5.4 CIT size

For our evaluation, we use a 512-entry table to identify critical instruction fetches. We perform a sweep across different table sizes, ranging from 256 to 2048 entries, and quantify ICARUS speedup as shown in Figure 5.11. We observe that increasing the table size beyond 2048 entries results in only marginal improvement(less than 0.01%) in overall performance, indicating diminishing returns. On the other hand, reducing the table size below 512 entries, such as to 256 entries, leads to a noticeable performance degradation due to insufficient tracking of critical instructions. It is important to note that the optimal size of the Criticality Indicator Table(CIT) depends on factors such as the instruction working set and the L2 cache size. A smaller L2 cache may result in a higher number of critical instructions due to increased cache misses at L2, while larger working sets can also increase the number of critical instructions that need to be tracked. Although a 512-entry table works well for our experimental setup, the CIT size is configurable and can be tuned based on system requirements and workload characteristics.

5.5.5 Cache hierarchies

So far, we have evaluated ICARUS on an Intel Xeon Granite Rapids-like cache hierarchy. To further validate its generality and robustness, we analyze ICARUS across additional modern

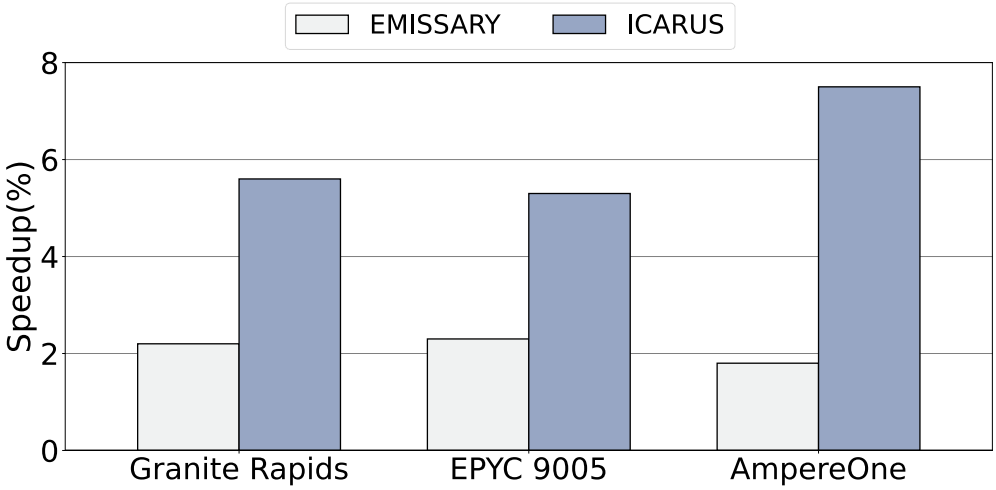


Figure 5.12: Sensitivity study: effect of cache hierarchies on performance

server cache hierarchies, including an AMD Zen 5 [13] based EPYC 9005(Turin) [40] processor and an ARM-based AmpereOne [41] processor. The detailed cache parameters for these architectures are shown in Table 5.3 and Table 5.4. Figure 5.12 presents the performance of EMISSARY and ICARUS across these three different cache hierarchies. Overall, ICARUS consistently outperforms EMISSARY, demonstrating its effectiveness across diverse architectural designs.

AMD EPYC 9005. The AMD EPYC 9005(Turin), based on the Zen 5 microarchitecture, represents a high-performance server processor with large cache hierarchies and high associativity. In particular, the L2 cache is 16-way associative, which aligns well with our design assumptions. Based on our study, the watermark thresholds for criticality and reuse-based bins(BRC) remain similar to our default configuration. Specifically, we use watermark values of two, four, six, and four for the four bins: [0,0], [0,1], [1,1], and [1,0], respectively, based on criticality and reuse bit. Additionally, we retain a 512-entry CIT structure, which continues to provide a good balance between hardware overhead and performance.

AmpereOne. We also evaluate ICARUS on the AmpereOne processor, which is designed as a power-efficient ARM-based server CPU targeting cloud and scale-out workloads. Compared to high-performance x86 processors, AmpereOne employs relatively lower associativity(8-way in L2), which impacts the granularity of replacement decisions. Accordingly, we adjust the watermark thresholds to one, two, three, and two for the bins [0,0], [0,1], [1,1], and [1,0] based on criticality and reuse bit, to better match the reduced associativity. Despite these differences, ICARUS continues to perform effectively, highlighting its adaptability across

Table 5.3: EPYC 9005-like cache hierarchy

Field/Model	**AMD 9005-like cache hierarchy**
L1I/L1D caches	32KB(I)/48KB(D), 8/12 way, 64B cache line, 4 cycles
Unified L2	1MB, 16-way, 64B line size, 14 cycles
Shared L3	4MB, 16-way, 64B line size, 40 cycles

Table 5.4: AmpereOne-like cache hierarchy

Field/Model	**AmpereOne-like cache hierarchy**
L1I/L1D caches	16KB(I)/64KB(D), 4-way, 64B cache line, 3/4 cycles
Unified L2	2MB, 8-way, 64B line size, 11 cycles
Shared L3	1MB, 8-way, 64B line size, 10 cycles

architectures with varying design tradeoffs.

Performance. Quantitatively, ICARUS achieves an average speedup of 5.6% on Granite Rapids, 5.3% on EPYC 9005, and 7.5% on AmpereOne, compared to 2.2%, 2.3%, and 1.8% for EMISSARY, respectively. Notably, the gains on AmpereOne are higher, indicating that ICARUS is particularly effective in architectures with smaller caches having high frontend bottleneck. Overall, these results demonstrate that ICARUS is not tightly coupled to a specific cache hierarchy design. Instead, it generalizes well across processors with different cache sizes, associativity, etc, making it a practical solution for a wide range of modern datacenter and cloud systems.

5.6 ICARUS for applications with high critical instruction fetches and small L2 sizes

We observe that a large number of instruction fetches are critical for applications like `Verilator` with a smaller L2, such as 1MB L2 (same as AMD's Zen 5 [13]). Through empirical analysis, we observe that with `Verilator`, more than 50% of instruction fetches are critical with an extremely high L2 instruction MPKI of 61.52 if we use an L2 of 1MB. On the other hand, the data MPKI at the L2 is just 0.24. We observe that, with high L2 instruction MPKI, `Verilator` sees hits at L3, causing decode starvation for more than 40 cycles, which is a significant front-end bottleneck. With ICARUS for a 1MB L2 [13], a large fraction of L2 instruction lines becomes critical. However, there are a few L2 instruction lines that cause decode starvation for more than 40 cycles. So, for `Verilator`, there is a need to preserve highly costly critical instruction lines. To accommodate the same, we redefine

the definition of critical fetch (critical instruction line) as any instruction fetch that causes decode starvation for more than 40 cycles. In this way, ICARUS preserves costly critical instructions. We use a threshold based on the fraction of critical fetches to trigger ICARUS that preserves only the costly critical lines. In our case, the threshold is 8% (more than the average). Depending on the application behavior, both the cost and the threshold to identify applications with costly fetches can be tuned.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This dissertation proposed ICARUS, a front-end criticality and reuse-based L2 replacement policy prioritizing instruction lines for datacenter applications. We argued for context-based critical fetch detection and branch history to detect critical fetches in the presence of the dynamic behavior of critical fetches. Next, we proposed a criticality and reuse-based L2 replacement policy that preserves critical instruction lines with long reuse. On average, across 12 datacenter applications, ICARUS outperforms the baseline Tree-based Pseudo LRU policy by 5.6% with a maximum improvement of 51%, which is a significant improvement compared to the state-of-the-art replacement policy EMISSARY that improves performance by 2.2% over the Tree-based Pseudo LRU policy. ICARUS provides this performance improvement with a storage overhead of 8.13KB, which is marginal compared to a 2MB L2. We further demonstrated the robustness of ICARUS by evaluating it across a wide range of microarchitectural configurations, including different L1I, L2, and BTB sizes. Additionally, we validated the effectiveness of ICARUS across multiple modern datacenter processor cache hierarchies, including Intel, AMD EPYC, and ARM-based AmpereOne architectures, highlighting its adaptability to diverse cache hierarchies and design tradeoffs. Overall, ICARUS provides a practical and scalable solution for improving front-end performance in modern datacenter processors.

6.2 Future Work

While ICARUS demonstrates significant improvements in front-end performance, there are several promising directions for future work. First, the notion of criticality in ICARUS, derived using branch history as context information, can be extended beyond cache replacement. In particular, this criticality signal can be leveraged to design more effective Branch Target Buffer (BTB) replacement policies. By prioritizing entries associated with critical control-flow instructions, future designs can further reduce front-end stalls. Second, ICARUS combines branch history and decode starvation to identify critical instruction fetches. This information can be integrated with instruction prefetching mechanisms to proactively fetch critical cache lines. In particular, predicting long-reuse critical instruction lines and prefetching them ahead of time can help mitigate cache misses and further reduce front-end latency. Finally, the current design of ICARUS focuses solely on instruction criticality, prioritizing L2 instruction cache lines. While this improves front-end performance, it may increase data MPKI due to contention in shared L2 among instruction and data. Therefore, an important direction for future work is to incorporate data criticality and explore the synergy between instruction and data criticality. Overall, extending ICARUS along these directions can further enhance its applicability and effectiveness in next-generation datacenter processors.

Chapter 7

Acknowledgements

Firstly, I express my gratitude to my advisor, **Prof. Biswabandan Panda**, for his support and invaluable guidance throughout my journey in this institution. His expertise, patience, and encouragement have been instrumental in shaping this research project and my academic growth.

I extend my appreciation to **Prof. Alberto Ros** for his collaboration on this work. His contribution, expertise, and insightful suggestions have immensely enriched the research and added depth to our findings.

I would also like to thank my peer **Hrishikesh** for all his contributions and for helping me throughout the research work.

Vedant Uddhaorao Kalbande
IIT Bombay

Bibliography

[1] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 462–473.

[2] M. Ugur, C. Jiang, A. Erf, T. Ahmed Khan, and B. Kasikci, "One profile fits all: Profile-guided linux kernel optimizations for data center applications," *SIGOPS Oper. Syst. Rev.*, vol. 56, no. 1, p. 26–33, Jun. 2022. [Online]. Available: https://doi.org/10.1145/3544497.3544502

[3] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 158–169.

[4] "Apache kafka," https://kafka.apache.org/, 2023.

[5] "Spec cpu 2017," https://www.spec.org/cpu2017/, 2017.

[6] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Seznec, D. Tullsen, and D. I. August, "Pdip: Priority directed instruction prefetching," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 846–861.

[7] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher," *The 1st Instruction Prefetching Championship (IPC1)*, 2020.

[8] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 99–111.

[9] T. Zhang, B. Grot, W. He, Y. Lv, P. Qu, F. Su, W. Wang, G. Zhang, X. Zhang, and Y. Zhang, "Hierarchical prefetching: A software-hardware instruction prefetcher for server applications," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. New York, NY, USA: Association for Computing Machinery, 2025, p. 529–544.

[10] D. Chasapis, G. Vavouliotis, D. A. Jiménez, and M. Casas, "Instruction-aware cooperative tlb and cache replacement policies," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 619–636. [Online]. Available: https://doi.org/10.1145/3669940.3707247

[11] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 60–71, 2010.

[12] N. P. Nagendra, B. R. Godala, I. Chaturvedi, A. Patel, S. Kanev, T. Moseley, J. Stark, G. A. Pokam, S. Campanoni, and D. I. August, "Emissary: Enhanced miss awareness replacement policy for l2 instruction caching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–13.

[13] "Amd zen5," https://en.wikipedia.org/wiki/Zen_5, 2024.

[14] "Arm neoverse v2," https://chipsandcheese.com/p/hot-chips-2023-arms-neoverse-v2/, 2023.

[15] "Intel granite rapids," https://en.wikipedia.org/wiki/Granite_Rapids, 2024.

[16] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 16–27.

[17] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 172–182.

[18] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of

exclusive caches," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 343–353.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[20] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[21] A. Seznec, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016. [Online]. Available: https://jilp.org/jwac-2/program/cbp3_07_seznec.pdf

[22] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 519–532.

[23] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 734–747.

[24] Y. Wang, C.-H. Chang, A. Sivasubramaniam, and N. Soundararajan, "Acic: Admission-controlled instruction cache," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 165–178.

[25] S. Hines, D. Whalley, and G. Tyson, "Guaranteeing hits to improve the efficiency of a small instruction cache," in *40th annual IEEE/ACM international symposium on microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 433–444.

[26] J. Kin, M. Gupta, and W. H. Mangione-Smith, "The filter cache: An energy efficient memory structure," in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 184–193.

[27] D. Chasapis, G. Vavouliotis, D. A. Jiménez, and M. Casas, "Instruction-aware cooperative tlb and cache replacement policies," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. Association for Computing Machinery, 2025, p. 619–636. [Online]. Available: https://doi.org/10.1145/3669940.3707247

[28] S. Oh, M. Xu, T. A. Khan, B. Kasikci, and H. Litz, "UDP: Utility-Driven Fetch Directed Instruction Prefetching ," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Jul. 2024, pp. 1188–1201. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISCA59077.2024.00089

[29] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 99–111.

[30] S. Song, T. A. Khan, S. M. Shahri, A. Sriraman, N. K. Soundararajan, S. Subramoney, D. A. Jiménez, H. Litz, and B. Kasikci, "Thermometer: profile-guided btb replacement for data center applications," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 742–756. [Online]. Available: https://doi.org/10.1145/3470496.3527430

[31] K. Zhu, Y. Zhao, Y. Gao, P. Braun, T. A. Khan, H. Litz, B. Kasikci, and S. Deng, "From optimal to practical: Efficient micro-op cache replacement policies for data center applications," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 716–731.

[32] B. Panda, "CLIP: load criticality based data prefetching for bandwidth-constrained many-core systems," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 714–727.

[33] H. Litz, G. Ayers, and P. Ranganathan, "CRISP: critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. Association for Computing Machinery, 2022, p. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

[34] A. Deshmukh and Y. N. Patt, "Criticality driven fetch," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. Association for Computing Machinery, 2021, p. 380–391. [Online]. Available: https://doi.org/10.1145/3466752.3480115

[35] S. S. Kim and A. Ros, "Effective context-sensitive memory dependence prediction," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 515–527.

[36] S. McFarling, "Combining branch predictors," in *Vol. 49. Technical Report TN-36, Digital Western Research Laboratory, 1993*, 1993.

[37] "Gem5-emissary," https://github.com/PrincetonUniversity/gem5_FDIP.git, 2023.

[38] B. R. Godala, "Arm 64-bit datacenter workloads used in emissary paper," https://drive.google.com/file/d/1ac60R-nuENQjw-rRBR-0S9rYQEEuCvyp/view, 2024, accessed: 2024-12-07.

[39] A. Seznec, "A 64-Kbytes ITTAGE indirect branch predictor," in *JWAC-2: Championship Branch Prediction*. San Jose, United States: JILP, Jun. 2011. [Online]. Available: https://jilp.org/jwac-2/program/cbp3_07_seznec.pdf

[40] "Amd epyc 9005(turin)," https://chipsandcheese.com/p/amds-turin-5th-gen-epyc-launched, 2024.

[41] "Ampereone," https://chipsandcheese.com/p/ampereone-at-hot-chips-2024-maximizing-density, 2024.